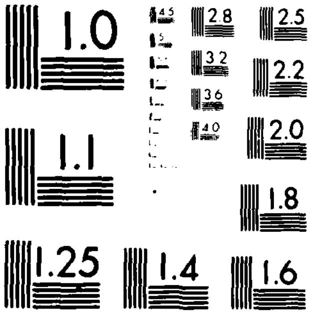
AD-A109 978 TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP F/G 9/2
ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPE--ETC(U)
DEC 81 F30602-80-C-0293
UNCLASSIFIED RADC-TR-81-360-VOL-3 NL

1 OF 3
AD
A109978

A109 9



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

PHOTOGRAPH THIS SHEET

AD A109978

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

[redacted]

Texas Instruments, Inc
Lewisville, TX Equipment Group - ACSL
ADA Integrated Environment III Computer Program Development Specification. Interim Rept. 15 Sep. 80 - 15 Mar. 81
Contract F30602-80-C-0293 RADC-TR-81-360, Vol. III Dec. '81

DOCUMENT IDENTIFICATION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR
NTIS GRA&I ☒
DTIC TAB □
UNANNOUNCED □
JUSTIFICATION

BY
DISTRIBUTION /
AVAILABILITY CODES

DIST	AVAIL AND/OR SPECIAL	
A		

DISTRIBUTION STAMP

DTIC COPY INSPECTED 2

DTIC
SELECTED
JAN 25 1982
D

DATE ACCESSIONED

82 01 12 008

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

RADC-TR-81-360, Vol III (of four)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Texas Instruments, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-360, Volume III (of four) has been reviewed and is approved for publication.

APPROVED: Elizabeth S. Kean

ELIZABETH S. KEAN
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-360, Vol III (of four) 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPECIFICATION	5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81 6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0293
9. PERFORMING ORGANIZATION NAME AND ADDRESS Texas Instruments Incorporated Equipment Group-ACSL, P O Box 405, M.S. 3407 Lewisville TX 75067	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F/62702F 55811919
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441	12. REPORT DATE December 1981 13. NUMBER OF PAGES 180
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same	
18. SUPPLEMENTARY NOTES RADC Project Engineer: Elizabeth S. Kean (COES)	

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE of CONTENTS

Paragraph Title Page

SECTION 1 SCOPE

1.1 Identification 1-1
1.2 Functional Summary 1-1

SECTION 2 APPLICABLE DOCUMENTS

2.1 Program Definition Documents 2-1
2.2 Military Specifications and Standards 2-1

SECTION 3 REQUIREMENTS

3.1 Introduction 3-1
3.1.1 General Description 3-1
3.1.1.1 The Front End 3-2
3.1.1.2 The Back End 3-2
3.1.1.3 Compiler Control File 3-3
3.1.1.4 Automatic Generation of Pathnames 3-5
3.1.1.5 Error Reporting 3-5
3.1.1.6 Generation of Listings 3-6
3.1.1.6.1 Source Listing and Error Messages 3-6
3.1.1.6.2 Symbol Table Listing 3-7
3.1.1.6.3 Symbol Cross Reference Listing 3-7
3.1.1.6.4 Environment Listing 3-8
3.1.1.6.5 IL Listing 3-8
3.1.1.6.6 Object Code Listing 3-8
3.1.1.7 Bootstrapping the Compiler 3-9
3.1.1.8 Retargeting and Rehosting the Compiler 3-11
3.1.2 Program Interfaces 3-11
3.1.2.1 Interface with the Library File Utility 3-12
3.1.2.2 Interface with the Program Binder 3-12
3.1.2.3 Interface with the Database Manager 3-12
3.1.2.4 Interface with the Source Level Debugger . . . 3-12
3.1.2.5 Interface with Listing Tools 3-12
3.1.3 Function Descriptions 3-13
3.1.3.1 The Analyzer 3-13

3.1.3.2 The Expander/Optimizer 3-13
3.1.3.3 The Code Generator 3-14
3.2 Detailed Functional Requirements 3-14
3.2.1 The Analyzer 3-14
3.2.1.1 Inputs 3-15
3.2.1.2 Initialization 3-15
3.2.1.3 Lexical Analysis 3-17
3.2.1.4 Syntax and Semantic Analysis 3-19
3.2.1.5 Semantic Analysis Issues 3-23
3.2.1.5.1 Enforce Visibility Rules 3-23
3.2.1.5.2 Static Expression Evaluation 3-24
3.2.1.5.3 Error Handling 3-24
3.2.1.5.4 Generics 3-24
3.2.1.5.4.1 Processing a Generic Declaration and Body . 3-25
3.2.1.5.4.2 Processing a Generic Instantiation 3-26
3.2.1.5.4.3 Optimization of Generic Instantiations . . 3-27
3.2.1.5.5 The INLINE Pragma 3-27
3.2.1.5.6 Representation Specifications 3-28
3.2.1.6 Saving the IL 3-28
3.2.1.7 Compilation Statistics 3-29
3.2.1.8 Formation of the Statement Map 3-30
3.2.1.9 Outputs 3-30
3.2.2 The Expander/Optimizer: Introduction 3-31
3.2.2.1 Inputs 3-32
3.2.2.2 Pre-processing 3-32
3.2.2.3 Expansion/Local Optimization 3-33
3.2.2.3.1 Representational Decisions 3-33
3.2.2.3.2 Expansion of the Abstract Syntax Tree 3-33
3.2.2.3.2.1 Inline Subprogram Expansion 3-33
3.2.2.3.2.2 Expansion of a Generic Instantiation . . . 3-35
3.2.2.3.2.3 Other Tree Transformations 3-36
3.2.2.3.3 Local Optimization 3-37
3.2.2.4 Global Optimization 3-38
3.2.2.5 Saving the Expanded/Optimized IL 3-38
3.2.2.6 Outputs 3-38
3.2.3 The Code Generator: Introduction 3-39
3.2.3.1 Inputs 3-40
3.2.3.2 Pre-processing 3-40
3.2.3.3 Storage Allocation 3-41
3.2.3.4 Code Generation 3-42
3.2.3.4.1 Code Generation Schema 3-43
3.2.3.4.2 Processing Representation Specifications . . 3-43
3.2.3.4.3 Processing Language Attributes 3-44
3.2.3.4.4 Processing Global References 3-44
3.2.3.5 Post-Processing (Final Optimizations) 3-44
3.2.3.5.1 Branch and Label Optimization 3-45
3.2.3.5.2 Peephole Optimization 3-46
3.2.3.5.3 Cross-Jumping 3-48
3.2.3.6 Formation of Object Module 3-48
3.2.3.7 Formation of Symbol Information for the Source

Level Debugger 3-48
3.2.3.7.1 The Symbol Map 3-49
3.2.3.7.2 The Type Map 3-49
3.2.3.8 Updating the Statement Map 3-49
3.2.3.9 Outputs 3-50

SECTION 4 QUALITY ASSURANCE PROVISIONS

4.1 Introduction 4-1
4.1.1 Computer Program Component Test and Evaluation . 4-1
4.1.2 Integration Testing 4-2
4.1.3 Formal Acceptance Testing 4-2
4.2 Test Requirements 4-2
4.2.1 Bootstrap tests 4-2
4.2.2 Ada Compiler Validation Tests 4-3
4.2.3 Rehosting tests 4-3
4.3 Acceptance Test Requirements 4-3
4.3.1 Performance Requirements 4-3
4.4 Independent Validation and Verification 4-3

APPENDIX A THE COMPILER CONTROL LANGUAGE

A.1 Introduction A-1
A.2 Command Language Syntax A-1
A.3 Control Language Semantics A-2
A.3.1 The LIBRARY Command A-2
A.3.2 The UNIT Command A-2
A.3.3 The PRAGMA Command A-3
A.3.4 The NAME Command A-3
A.3.5 The P_UNIT Command A-3
A.4 Command Stream Examples A-4

APPENDIX B A RECURSIVE DESCENT PARSER FOR ADA

B.1 Construction of the Recursive Descent Parser . . . B-1
B.1.1 The Basic Approach B-1
B.1.2 Left Factoring B-4
B.1.3 Composition B-5
B.1.4 Combination of Non-terminal Productions B-6
B.1.5 Incorporating Semantics B-7
B.1.6 Elimination of Left Recursion B-7

APPENDIX C A PRACTICAL GRAMMAR FOR ADA

C.1 Ada Concrete Syntax C-1
C.2 Productions C-10

APPENDIX D THE SYMBOL DICTIONARY

D.1 Underlying Data Structures D-1
D.1.1 The Visibility Stack D-3
D.1.2 Declared Symbol Tables D-4
D.1.2.1 Block, Subprogram, Package and Task Declarators D-5
D.1.2.2 Symbol Table Entries and Fields D-7
D.1.2.3 Type Table Entries and Fields D-9
D.1.2.4 Virtual Symbol Table D-11
D.1.3 The Look-Up Table D-11
D.1.4 The Name Table D-13
D.2 The Predefined Environment D-13
D.2.1 The Package STANDARD D-13
D.2.2 Reserved Words and Pragmas D-14
D.3 Maintaining the Compilation Context D-15
D.3.1 Basic Block Structure Operations D-15
D.3.2 Processing Packages D-25
D.3.3 Processing WITH and USE Clauses D-29
D.4 Enforcing the Visibility Rules D-31
D.4.1 Direct Visibility and Hidden Identifiers D-31
D.4.2 Loop Parameters D-32
D.4.3 Overloading and Operator Identification D-32
D.4.4 Selected Components D-34
D.4.5 Direct Visibility for Package Entities via USE Clause D-36
D.5 The RENAME Statement D-38
D.6 Derived Types D-40

APPENDIX E REPRESENTATIONS OF DIANA

E.1 Design Issues E-1
E.2 Source Position E-2
E.3 Internal Representation E-3
E.4 External Representation E-4
E.4.1 Naming Nodes E-4
E.4.2 Text versus Binary Representation E-4
E.4.3 Representation of DIANA on Files E-5
E.4.4 Transforming DIANA on a File to the Internal Representation E-6
E.4.4.1 Constructing a Symbol Table E-6

E.4.4.2 Constructing an AST E-7
E.5 Visible Representation E-8
E.6 Derivation of Dialects E-8

APPENDIX F THE LIBRARY FILE UTILITY

F.1 Introduction F-1
F.2 The Library File F-1
F.2.1 The Library File Descriptor F-2
F.2.2 Compilation Unit Name Table F-2
F.2.3 The Compilation Unit Descriptor F-3
F.3 Functional Capabilities F-6
F.4 Library File Interface F-6

APPENDIX G REFERENCES

LIST of TABLES

Table	Title	Page
3-1	Lexical Unit Constructs	3-17
3-2	Outputs of the Analyzer	3-31
3-3	Machine-Independent Optimizations	3-32
3-4	Outputs of the Code Generator	3-50
A-1	Output File Designators	A-3
D-1	USE Visibility Determination Table--Entering Scope .	D-36
D-2	USE Visibility Determination Table--Leaving Scope .	D-38

LIST of FIGURES

Figure	Title	Page
3-1	Ada Compiler's Structure	3-1
3-2	Structure of the Front End	3-2
3-3	Structure of the Back End.	3-3
3-4	Structure of the IBM 370 Bootstrap Compiler.	3-9
3-5	Compiler Environment Interfaces.	3-11
3-6	Structure of the Analyzer.	3-15
3-7	Organization of the Front End.	3-22
3-8	Generic Definition and Instantiation Data Flow . . .	3-26
3-9	Generic Expansion Data Flow.	3-36
3-10	Representation of a Code Node.	3-45
A-1	Compiler Command Language Syntax	A-2
D-1	Organization of the Symbol Dictionary.	D-2
D-2	Structure of the Visibility Stack.	D-3
D-3	Visibility Stack Entry	D-4
D-4	Organization of Declared Symbol Tables	D-5
D-5	Block, Subprogram, Package and Task Declarator . . .	D-6
D-6	Symbol Table Entry	D-8
D-7	Type Table Entry	D-10
D-8	Look-Up Table and Symbol Table Interface	D-12
D-9	Initialization of the Symbol Dictionary with STANDARD	D-14
D-10	Typical Block Structure.	D-17
D-11	Enter 'P'.	D-18
D-12	Enter 'Q'.	D-19
D-13	Leave 'Q', Enter 'R'	D-20
D-14	Enter 'S'.	D-21
D-15	Leave 'S', Enter'T'.	D-22
D-16	Leave 'T', Still in 'R'.	D-23
D-17	Leave 'R', Still in 'P'.	D-24
D-18	Leave 'P', Still in Parent of 'P'.	D-25
D-19	Typical Block Structure With A Package	D-26
D-20	In Package 'Q', Leaving 'Q1', Entering 'Q2'.	D-27
D-21	In Proc 'P', Leaving 'Q', Entering 'R'	D-28
D-22	In Proc 'S', 'Use Q'	D-29
D-23	WITH Package ABC	D-30
D-24	USE Package ABC.	D-31
D-25	Overloading Chains in the Symbol Table	D-33
D-26	Operator Identification Structures	D-34
D-27	Selected Component Identification Algorithm.	D-35
D-28	Newname RENAMES Oldname.	D-40
D-29	Linkages for a Derived Type.	D-41
E-1	Representations of DIANA	E-1
F-1	Interface with the Library File Utility	F-7

LIST of EXAMPLES

Example	Title	Page
A-1	Compiler Command Stream for the Analyzer	A-4
A-2	Compiler Command Stream for the Expander/Optimizer .	A-4
A-3	Compiler Command Stream for the Code Generator . . .	A-5

SECTION 1

SCOPE

1.1 Identification

This specification establishes the requirements for the performance, design, test and acceptance of a computer program identified as the Ada optimizing compiler. This CPCI provides a programmer or test engineer with the ability to compile Ada compilation units. The compiler is hosted on the IBM 370 operating under the Ada Integrated Environment (AIE), or on the Interdata 8/32 under the Ada Integrated Environment.

1.2 Functional Summary

The purpose of this specification is two-fold:

1. To identify the functional capabilities of the Ada optimizing compiler

2. To describe the interface between the Ada optimizing compiler and the AIE, the command language, the library file, and the program binder

The Ada optimizing compiler is a compiler for the Ada language as defined by the Language Reference Manual [DoD80B]. It consists of four passes. The first pass (front end) performs lexical analysis, syntactic analysis, and static semantic analysis, and produces the intermediate language, DIANA. The next pass expands the high-level intermediate language into a lower semantic level dialect which includes machine dependent information. A (logically) optional optimization pass performs machine independent optimizations on the intermediate language. A code generation phase then converts the dialect of DIANA output by the expansion or optimization pass to target machine code. The entire system is written to be as transportable and machine relative as is practically possible.

SECTION 2

APPLICABLE DOCUMENTS

2.1 Program Definition Documents

[DoD80A] Requirements for Ada Programming Support Environments: "STONEMAN", DoD (February 1980).

[RADC80] Revised Statement of Work for Ada Integrated Environemnts, RADC, Griffiss Air Force Base, NY (March 1980).

[SOFT80A] Ada Compiler Validation Capability: Long Range Plan, SofTech Inc., Waltham, MA (February 1980).

[SOFT80B] Draft Ada Compiler Validation Implementers' Guide, SofTech Inc., Waltham, MA (October 1980).

2.2 Military Specifications and Standards

[DoD80B] Reference Manual for the Ada Programming Language: Proposed Standard Document, DoD (July 1980) (reprinted November 1980).

SECTION 3

REQUIREMENTS

3.1 Introduction

3.1.1 General Description

The compiler is divided into two parts: the machine independent front end and the machine dependent back end (cf. Figure 3-1).

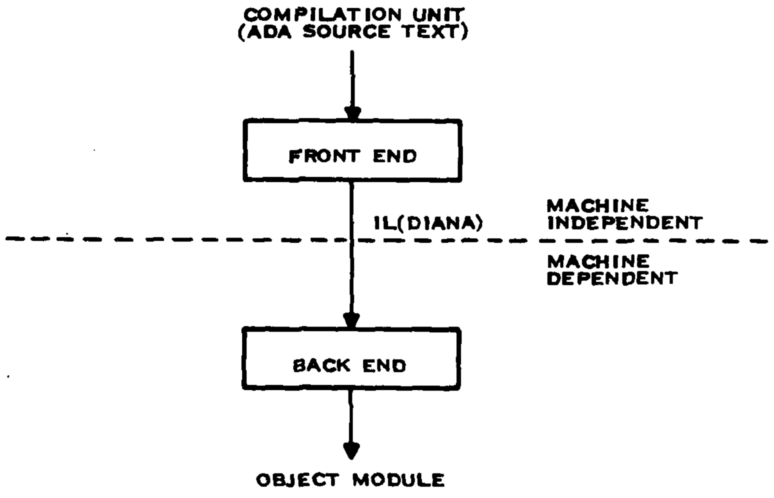


Figure 3-1 Ada Compiler's Structure

The front end (analyzer) and the back end (expander/optimizer and code generator) consist of separate, individually invoked tools. A system command language procedure can be written to invoke these tools sequentially, i.e., to compile a compilation unit.

3.1.1.1 The Front End

The front end, also known as the analyzer, is a language translator which accepts the Ada source text for a compilation unit, performs lexical analysis, checks the syntax and semantics of the compilation unit, and produces an intermediate representation/language of the compilation unit more convenient for later processing, i.e., optimization and code generation (cf. Figure 3-2). DIANA [GOO81] shall be used as the intermediate language (IL) (cf. Appendix E). The DIANA dialect output by the front end is a dialect common to all tools requiring IL as input.

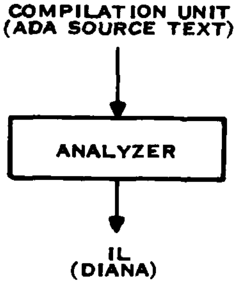


Figure 3-2 Structure of the Front End

3.1.1.2 The Back End

The back end is a language translator which accepts the DIANA dialect produced by the analyzer for a compilation unit and produces an object module for the compilation unit. The back end consists of two separate tools, i.e., the expander/optimizer, and the code generator (cf. Figure 3-3). In the expander/optimizer pass, the DIANA dialect output by the front end is expanded into a low-level machine dependent dialect and high payoff machine-independent optimizations are applied to the new dialect. Expansion of the high-level machine independent DIANA dialect produced by the front end into a low-level machine dependent DIANA dialect exposes the details of the computations and permits optimizations to be applied to them. The dialect also includes decisions on the run-time representation of data types and control constructs, thereby eliminating some tasks normally associated with a code generator. In order to perform machine independent optimization, pertinent information is assembled into usable constructs, such as flow graphs. After all expansions have been performed, local machine-independent optimizations with a high payoff are performed. Then, potentially low payoff, time consuming global machine independent optimizations specified by the implementation defined pragma OPTLEVEL are performed. If the pragma is present in a declarative part, the designated optimizations are applied to the body or block enclosing the declarative part.

The code generator accepts the DIANA dialect produced by the expander/optimizer, generates machine code for the target machine, and performs machine dependent optimizations. Its output is an object module for the compilation unit serving as the input to the program binder.

Each compiler pass is a tool which must be invoked in the proper order via the command language. The associated bookkeeping is maintained in the library file via the library file utility (cf. Appendix F).

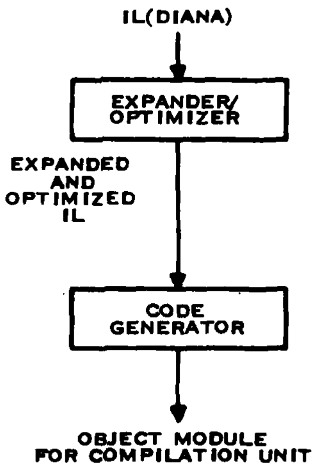


Figure 3-3 Structure of the Back End

3.1.1.3 Compiler Control File

Input to each pass of the compiler is the pathname of a control file specifying its input parameters. The control file contains a command stream expressed in the compiler control language (cf. Appendix A). The command stream specifies, for one or more compiler passes, a block of control sequences. The block of control sequences is delineated by a labeled BEGIN-END pair; the label designating the pass. It is the function of each compiler pass to scan the control file for its block of control sequences. Each control sequence designates and applies to a compilation unit to be processed.

The input parameters to a compiler pass consists of:

- the pathname of a library file
- a list of compilation units to be processed
- language pragmas
- pathnames of output files

The pathname of a library file must be specified first since all compilation bookkeeping for a program library is maintained in it. The library file must be created initially via a call to the creation subprogram in the library file utility (cf. Appendix F).

The nature of the list of compilation units depends on the compiler phase. For the analyzer, an item in the list is the pathname of an Ada source text file. This source file may contain one or more compilation units. For the expander/optimizer and code generator passes, an item in the list is the pathname of the appropriate input IL file of the compilation unit being processed.

Pragmas may be specified that override or augment those specified in the source text of a compilation unit. It is as if the pragma had occurred in the position specified in the pragma's definition (except for LIST which is positioned before the named unit). Since a pragma's extent is confined to the compilation unit in or before which it appears, it is not necessary to construct for a compilation unit a set of in-scope pragmas (by examining the DIANA of other compilation units according to the static structure of the program as maintained in the library file). Pragmas, in a control sequence, permit overriding pragmas specified in the source text without having to modify the source text and recompiling it. (Since the overriding pragma is recorded in the control file which is part of the derivation history of the output, the output object in the database can be re-derived.) This flexibility permits, for example, compilation units to be compiled and debugged without optimization during program development and then "recompiled" with optimization for the production version. Recompilation consists only of reprocessing the DIANA. Therefore, other compilation units need not be recompiler as would be required when source text is changed and recompiled.

Pathnames of output files may be specified in a control sequence; otherwise, they are assigned by default (cf. Section 3.1.1.4).

3.1.1.4 Automatic Generation of Pathnames

The pathnames of IL, error, debugger information and object files are assigned by default by the compiler unless a specific pathname is given in the control file. (The same naming conventions are also followed by the listing tools for listing file pathnames.) The format of the pathname is:

```
library_file.library_unit{.subunit}.category
```

The 'category' is defined in the output section for each compiler phase. The 'category' is the concatenation of the database category attribute and a signature. The signature is a unique integer assigned by the library file utility (cf. Appendix F). It uniquely identifies a particular file in a given category when there are more than one.

The 'library_file' portion of the pathname is given as input in the control file. The 'library_unit{.subunit}' portion is derived as follows:

* For the front end compiler pass, it is generated from the name of the compilation unit. If the compilation unit is a subunit, it is derived from the name of the subunit and the name of its parent unit.

* For the back end passes, a consistency check is first made. The input IL file's dictionary (cf. Section E.4.3) contains a unique integer index used to reference the entry for the corresponding compilation unit in a the library file and the database name of the library file (cf. Appendix F). The library file pathname given as input in the control file is mapped to its database name and checked against the database name of the library file given in the input IL file's dictionary. The two must match. The name of the compilation unit (i.e., library_unit{.subunit}) is obtained from the compilation unit's entry in the library file.

3.1.1.5 Error Reporting

The compiler detects all syntax errors and all semantic errors as specified in the Ada Reference Manual [DoD80B]. Errors detected by a compiler pass are output to an error file associated with an IL or object file output by the pass. The default pathname of the error file is:

```
library_file.library_unit{.subunit}.ERROR
```

The pathname is constructed from information contained in the entry of the compilaton unit in the library file. An error file contains a sorted list of error descriptor. An error descriptor consists of an error number, a severity code, an index to an entry in the Source Table (contained in the IL file for the compilation unit (cf. Appendix E, Section E.4.3) containing the databasename of the source file, the line number of the text where the error occurred, and the character position within the line. Error numbers are unique for a compiler pass.

The severity code indicates the correctness of the resulting output of a compiler pass. The severity codes are:

- Note (N): Information to the user about the compilation; compilation continues and the output is not affected.
- Warning (W): Information about the reliability or validity of the input; compilation continues and the output is not affected. The program may behave incorrectly at run-time.
- Error(E): Identification of an illegal syntactic or semantic construct. Reasonable recovery actions are enacted and compilation continues. The program may behave incorrectly or meaninglessly at run-time.
- Serious error (S): Identification of an illegal construct with no well-defined recovery action. The analyzer pass continues by deleting the DIANA for the construct. The expander/optimizer and code generator pass will not process the DIANA.
- Fatal error (F): Identificaton of an unrecoverable host environment error; compilation terminates and all opened files are properly closed.

3.1.1.6 Generation of Listings

Compiler output listings are generated by separate tools. With this approach, only the database names of the files used to generate the listings need to be recorded in the library file. The derivation of a listing file is maintained in database relations. Input parameters to the tool are the specificaion of a compilation unit and the pathname of the library file it is in. For the source listing, the pathname of a source file designates the compilation unit; for all other listings, the pathname of an IL file designates the compilation unit.

3.1.1.6.1 Source Listing and Error Messages

A source listing for a compilation unit may be generated via the tool LISTER from the original source text. A line consists of a line number, a statement number, and the text of the line. Lines and statements are numbered uniquely starting at 1; lines of INCLUDE files are numbered sequentially starting with the line number of the INCLUDE. Statement numbers are generated as the text is processed using the same subprograms used in the analyzer. The listing can be turned on/off via the language pragma LIST. INCLUDE pragmas are expanded unless, the listing is turned off.

Error messages from the compiler passes are merged into the source text as the source listing is generated. Error messages are placed under the the source line containing the error, or after the first available line following the source line containing the error. The line number is obtained from the error descriptor. Line numbers within an INCLUDE file must be decremented by the line number of the INCLUDE pragma; this base is kept in the Source Table. The error message consists of an error identifer, a severity code, a brief description of the error, and a ' ' underneath the token being processed at the time the error occurred. The description of the error is obtained from a table using the error number. The position of the ' ' is obtained from the error descriptor. Multiple errors for a source line are printed one error message per line in the order of the character position of the error, i.e., the errors associated with a line are sorted. An error identifer encodes which compiler pass generated it.

Each page of the listing contains the page number, the date and time the listing was produced, column numbers of 0 through 80 across the top of the page, the name of the compilation unit and the version of the compiler used to compile the compilation unit.

The default pathname of the listing file is:

```
library_file.library_unit{.subunit}.LIST.SOURCE
```

The pathname is constructed from information contained in the entry of the compilaton unit in the library file.

3.1.1.6.2 Symbol Table Listing

A listing of the symbol table for a compilation unit may be generated via the tool LST_SYMTAB. The listing consists of an alphanumeric list (in the collating sequence defined in the Ada Reference Manual [DoD80B]) of all symbols defined in the compilation unit and their charactertics. The symbol tables for all embedded program units and blocks are also generated. This information is useful both to the user and in managing and maintaining the compiler.

The default pathname of the symbol table listing file is:

```
library_file.library_unit{.subunit}.LIST.SYMTAB
```

The pathname is constructed from information contained in the entry of the compilation unit in the library file.

3.1.1.6.3 Symbol Cross Reference Listing

A symbol cross reference listing for a compilation unit may be generated via the tool LST_XREF. The listing consists of an alphanumeric list of all symbols defined in the compilation unit. With each symbol is its block level and a sorted list of statement numbers where the symbol is set and used. Statement numbers, where the symbol is set, are prefixed with an '*'.

The default pathname of the symbol cross reference listing file is:

```
library_file.library_unit{.subunit}.LIST.XREF
```

The pathname is constructed from information contained in the entry of the compilation unit in the library file.

3.1.1.6.4 Environment Listing

An environment listing for a compilation unit may be generated via the tool LST_ENVT. The listing consists of an alphanumeric ordered list of external names referenced by the compilation unit, e.g., names appearing in a USE clause, global variables and subprograms, etc. Associated with each name is the source position in the source where the external name is referenced.

The default pathname of the environment listing file is:

```
library_file.library_unit{.subunit}.LIST.ENVT
```

The pathname is constructed from information contained in the entry of the compilaton unit in the library file.

3.1.1.6.5 IL Listing

A listing of the visible representation of the generated DIANA (cf. Appendix E, Section E.5) may be generated via the tool LST_DIANA. The listing is especially useful in debugging and maintaining the compiler.

The default pathname of the DIANA listing file is:

```
library_file.library_unit{.subunit}.LIST.DIANA
```

The pathname is constructed from information contained in the entry of the compilaton unit in the library file.

3.1.1.6.6 Object Code Listing

A listing of the object code for a compilation unit may be generated by the tool LST_CODE. The listing contains one machine instruction per line: the machine code representation and the equivalent symbolic assembly language representation. A comment containing the source code statement number and statement shall precede the machine code for the statement. The assembly language makes correct use of all labels as they appear in the Ada source text; compiler generated labels are provided where no corresponding source label exists. Each page contains the name of the compilation unit, the time and date the code was produced, the compiler version, and the page number.

The default pathname of the code listing file is:

```
library_file.library_unit{.subunit}.LIST.CODE
```

The pathname is constructed from information contained in the entry of the compilation unit in the library file.

3.1.1.7 Bootstrapping the Compiler

Texas Instruments shall use the front end developed by the University of Karlsruhe for the German MoD to construct a bootstrap compiler. A preliminary release is expected in April-May 1981, with the final release scheduled for October 1981. The Karlsruhe front end accepts full Ada [DoD80B], outputs DIANA, the intermediate language preferred by Texas Instruments (cf. Appendix E), and implements the Ada separate compilation facility. It is a production quality front end and is available at no cost due to the Memorandum of Understanding between DoD and the German MoD. The bootstrap compiler permits implementation of the AIE tools to proceed in parallel, especially those whose interface is DIANA, and to be written in Ada from the outset.

The structure of the bootstrap compiler is given in Figure 3-4.

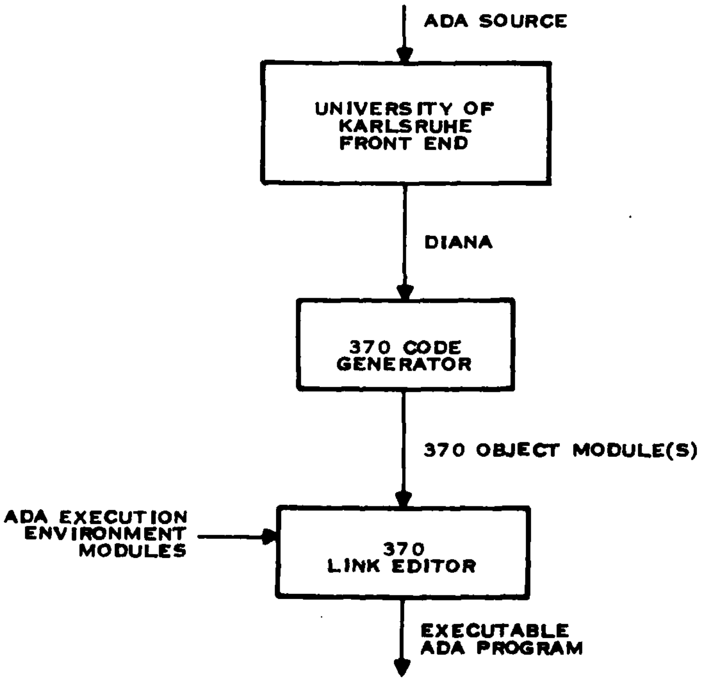


Figure 3-4 Structure of the IBM 370 Bootstrap Compiler

The front end was written in Ada-0, transformed to LIS, compiled by the Siemens 7760 LIS compiler, and linked with the LIS run-time package to produce an executable program [GOO80] which runs under the BS2000 operating system. The Siemens 7760 computer is essentially a copy of the IBM 370 and this fact permits the front end object modules to be transported to the IBM 370. Texas Instruments shall take the object modules constituting the executable front end and transform them into 370 object modules that may be linked and run on the IBM 370 under VM. This process involves (possibly) rewritting certain routines (e.g., the run-time initialization routine), emulating certain Siemens instructions (e.g., LBF, STBF), fixing the object format (e.g., ESD cards), and emulating certain OS functions (e.g., output).

Texas Instruments shall write a simple throw away IBM 370 code generator in Pascal that accepts Diana and outputs IBM 370 object modules. The code generator shall generate code based on the AIE Ada execution environment (cf. Ada Software Environment CPDS). The Ada execution environment for the bootstrap compiler shall be simplified and use only those features required by the subset of Ada needed to write the AIE compiler. The subset shall be rich enough to write and maintain the AIE compiler; the subset shall be similar to Ada-0 [GOO80] which does not include reals, tasking, and generics. The Ada execution environment shall be written in Ada and 370 assembly code. This approach permits Texas Instruments to become familiar enough with Diana to generate code and to verify some of its Ada execution environment concepts.

With these two executable programs, i.e., the front end and the 370 code generator, Ada programs can be compiled, linked with the Ada execution environment modules, and executed. In particular, the AIE tools can be compiled and executed in this manner.

To help describe the bootstrapping process, some definitions are needed. Deliverable compilers are discussed in terms of the host and target machines using the notation 'host-machine x target-machine', or for the case where the host and target machine are the same, simply the host machine name. For example, the products which results from this contract are the 370x370 compiler, the 370x8/32 compiler, and the 8/32x8/32 compiler. The first and last of these, at times, are referred to simply as the 370 compiler and the 8/32 compiler.

The first step of the bootstrap process shall be to develop the AIE 370 compiler. This shall be done using the bootstrap compiler. When the 370 compiler is producing valid code and passing other tests, this compiler shall be used to compile itself to produce the bootstrapped compiler. The process consists of compiling the 370 compiler three times. First, it is compiled using the bootstrap compiler. The output is then used to compile it again. The output of the second compilation is used to compile it a third time. The object produced by the second and third compilation are compared and must be identical. Once the compiler has reproduced itself in this manner, it is relatively stable.

3.1.1.8 Retargeting and Rehosting the Compiler

Having bootstrapped the 370 compiler, the process of retargetting the compiler to the Interdata 8/32 can begin. The process consists of re-parameterizing the machine-relative back end, i.e., the expander/optimizer and the code generator, and constructing the 8/32 code tables for the table-driven code generator. Since the 370 and 8/32 have similar architectures, the re-parameterization should be relatively straight forward. The result of compiling the 8/32 back end on the 370 is an executable 370 program that accepts DIANA and generates code for the 8/32.

The 8/32 back end is then tested and the generated code hand checked. Once the 8/32 code generator is generating proper 8/32 code on the 370, compiled code for small test programs shall be executed and tested on the 8/32. This requires the program binder and run-time support modules be retargetted for the 8/32. For the latter, this requires rewritting the low-level routines in 8/32 machine language and making algorithmic changes to those routines written in Ada. An executable 8/32 program is constructed by executing the retargetted 8/32 binder on the 370. Once cross-compiled programs are executing properly on the 8/32, the front end and the 8/32 back end compiler shall be compiled by the 370x8/32 compiler and the object modules bound. The resultant load module shall be transported to the 8/32 and tested. The 8/32 compiler shall be exercised and any remaining subtle errors corrected. Once it is executing properly on the 8/32 it shall be used to bootstrap itself. This is accomplished by compiling the source for the 8/32 compiler until it reproduces itself as described for the 370 system. When the project is finished, there shall be a compiler for the 370, for the Interdata 8/32, and a cross-compiler between the 370 and the 8/32.

3.1.2 Program Interfaces

The Ada optimizing compiler is an Ada program which interfaces with its environment through a number of files (cf. Figure 3-5). The unit of compilation is the compilation unit; the source text file may contain one or more compilaton units. The compilation state of each compilation unit constituting the program family is maintained in the library file. Each pass of the compiler is controlled by a control file. The major outputs of the compiler are object modules, information for the source level debugger, and information of the listing tools. There are a number of internal interfaces, i.e., DIANA dialects, between the different compiler passes which are described in the appropriate section of the pass (cf. Section 3.2.1, Section 3.2.2, Section 3.2.3).

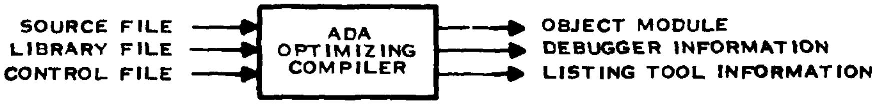


Figure 3-5 Compiler Environment Interfaces

The compiier interfaces with other Ada tools that perform requisite processing functions, i.e., the library file utility, the database manager, the source level debugger, and listing tools.

3.1.2.1 Interface with the Library File Utility

Th: library file utility provides the functional capabilities for maintaining the compilation state of a program contained in the program's library file. For each compilation unit in a program, there is an entry in the library file; the information associated with an entry is described in detai: in Appendix F. Information for a compilation unit is accessed (read or updated) via calls to subprograms.

3.1.2.2 Interface with the Program Binder

The library file is the interface between the compiler and the program binder. All information required to extract the object modules constituting a program is maintained in the library file by the compiler.

3.1.2.3 Interface with the Database Manager

The database manager provides the functional capabilities to create files, access file attributes, and establish a relationship between files. During the compilation process, the compiler generates a number of files (e.g., DIANA files, listing files, object module files) which are created by the database manager. The compiler maintains file attributes, especially those pertinent to it and which only it knows, e.g., the category of a file created by the compiler, the compiler identification, the compilation date/time. The compiler must also obtain values of file attributes, e.g., the category attribute to verify an input file. Finally, the compiler must also establish certain relationships between files resulting from the compilation, e.g., relate a DIANA and listing file with a source file, relate an object file with a DIANA file.

3.1.2.4 Interface with the Source Level Debugger

The information required by the Ada source level debugger is generated by the compiler and output to a number of files, viz., symbol tables (which are maintained in the DIANA files), symbol maps, type maps, and statement maps. Each of these files is described in detail in the section describing the compiler pass which generates them (cf. Section 3.2.1, Section 3.2.2, Section 3.2.3, Appendix E).

3.1.2.5 Interface with Listing Tools

The database names of the files, containing the information from which listings are generated, are contained with the entry for a compilation unit in a library file. This entry consists of the error files output by different compiler passes for the source listing; the DIANA files containing the symbol tables for the symbol table listing, the symbol cross reference listing, and the environment listing; the DIANA files containing the DIANA for the IL listing; and the object modules for the code listing.

3.1.3 Function Descriptions

The Ada optimizing compiler consists of three major passes: the analyzer, the expander/optimizer, and the code generator.

3.1.3.1 The Analyzer

The analyzer consists of four major tasks:

* The lexical analysis task converts Ada source text for a language construct into tokens and enters information about certain declared tokens (symbols) into a symbol table.

* The syntax analysis task checks if the tokens form a legal derivation based on the grammar using a recursive descent parsing algorithm and constructs an abstract syntax tree (AST) of the language structure.

* The semantic analysis task checks for valid language structures based on the static semantic rules of the language [C1180]. Generic instantiations and inline subprograms are expanded.

* The IL generator task writes out the IL (AST and symbol table) for the compilation unit in an external representation (cf. Appendix E, Section E.4.3) for use by other tools, in particular, the expander/optimizer and the code generator.

3.1.3.2 The Expander/Optimizer

This pass consists of the interleaving of IL expansion and local machine independent optimization followed by global machine independent optimization. Its major tasks are:

* Walk the abstract syntax tree and form the basic blocks and flow graph.

* Collect information about the unit by traversing its AST.

* Make run-time representational choices for data and control structures.

* Incorporate the representational choices explicitly into the high-level abstract syntax tree by expanding the AST.

* Perform local machine-independent optimizations on the expanded AST.

* Perform global machine-independent optimizations.

* Save the optimized IL.

3.1.3.3 The Code Generator

The major tasks of the code generator are:

* Map the machine characteristics onto the IL by transforming the IL into a lower-level IL more suitable for processing.

* Allocate storage (displacement in stack frames/heap, registers) for variables and literals in accordance with the abstract machine model and the characteristics of the target machine.

* Generate code for the target machine.

* Perform machine-dependent optimizations.

* Assemble code into an object module.

* Produce information required by Ada source level debugger.

3.2 Detailed Functional Requirements

3.2.1 The Analyzer

The purpose of the analyzer pass is to perform lexical, syntax, and static semantic analysis on one or more compilation units, and produce syntactically correct DIANA. The DIANA reflects the semantically analyzed compilation unit with type checking, overload resolution, and name binding performed. The structure of the analyzer (cf. Figure 3-6) is a consequence of using a recursive descent parser (see Appendix B for the rationale behind this approach).

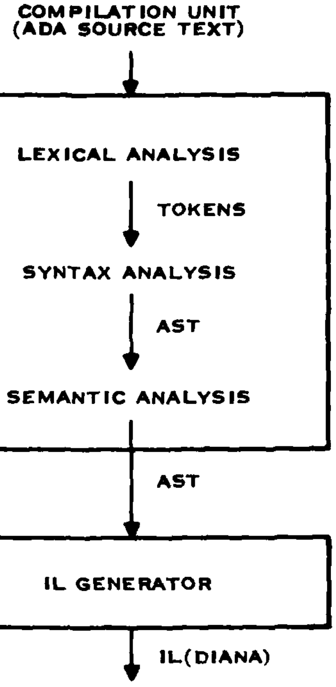


Figure 3-6 Structure of the Analyzer

3.2.1.1 Inputs

Input to the analyzer is the pathname of a control file (cf. Section 3.1.1.3). The control file contains the pathname(s) of source files for the compilation units to be processed. A source file may contain one or more compilation units. The control file must also specify the pathname of the library file in which the compilation state of the compilation unit(s) is maintained.

3.2.1.2 Initialization

Prior to processing the declaration or body part of a compilation unit, or the body of a subunit, the context specification and the name of the subunit's parent are scanned and a number of tasks related to separate compilation are performed, utilizing the library file utility and information contained in the library file (cf. Appendix F):

* Determine and validate the compilation context: From the library file, the genealogy of the compilation unit is found. This genealogy together with the merged list of library units occuring in WITH clauses constitute the compilation context. A unit mentioned in a WITH clause must be a library unit. For the qualified name of the parent of a subunit, the first component must be the name of a library unit that is the root of the genealogy,

and the remaining components must be subunits belonging to the genealogy of the library unit. The subunits must have been declared in one of the units or subunits of the genealogy.

* Ensure the proper order of compilation by performing the following checks:

- a compilation unit may only be compiled after all units mentioned in WITH clauses and/or a SEPARATE part are compiled.

- a package body must be be compiled after its specification.

- a subunit must be compiled after its parent unit.

* Create the compilation context for the compilation unit (cf. Appendix D): First, the symbol table for STANDARD is loaded. An implicit declaration of the named library units mentioned in the WITH clauses are entered in STANDARD's symbol table. Then, the symbol tables of compilation units elaborated in the USE clause of the compilation specification are made visible (cf. Appendix D). For a subunit, the symbol tables of all its ancestors, starting with parent unit, are loaded. If the compilation unit is the body of a separately compiled subprogram or package, i.e., its declaration part, the symbol table for that part is loaded (because it constitutes the local environment of the body)..

* Determine the compilation units requiring recompilation: Compilation units requiring recompilation are so marked in the library file. Compilation units requiring recompilation are determined as follows:

- When the specification of a subprogram is recompiled, the specification is checked against the previous specification. If the new specification is different, all calling units must be recompiled. If the specification is the same and the previous compilation contained an inline pragma for the subprogram but the current compilation does not, then all calling units for which the call was expanded inline must be recompiled.

- When the body of an inline subprogram is recompiled, all calling units for which the call was expanded inline must be recompiled.

- When the body of an inline subprogram is compiled or recompiled, a warning message is given listing all calling units for which the call was out-of-line and that should be recompiled.

3.2.1.3 Lexical Analysis

The principle task of the lexical analyzer or scanner is the classification of the source input into tokens. Each time a syntactical analyzer routine needs a new token, it calls the routine, SCAN, which will return the next token in the input stream. The token's value consists of an enumeration value and a pointer to the entry for the lexical unit, i.e., to either the symbol dictionary look-up table (cf. Appendix D, Section D.1.3) in the case of identifiers and operators, or to the name and string table (cf. Appendix D, Section D.1.4) for literals and strings. The enumeration value indicates the construct of the scanned lexical unit, such as reserved word, numeric literal or punctuation (cf. Table 3-1).

Table 3-1 Lexical Unit Constructs

Reserved Word
Identifier
Decimal Real
Decimal Integer
Based Number
Character Literal
Character String
Operator
Punctuation
Attribute

If an error should be detected by the lexical analyzer, the analyzer returns an error token so the syntax analyzer may be notified. The lexical analyzer then continues to scan from the point of error to the next delimiter so proper scanning can be continued.

The lexical scanner is also responsible for entering newly found identifiers and literal strings, both numeric and alphabetic, into the name and literal table and, when a new identifier is recognized, the look-up table.

Scanning is accomplished through the use of finite-state automatons. Techniques used will be a standard approach such as described by Aho [AHO77] and Dedourek [DED80]. Three such automatons are used to recognize all constructs required by the Ada Language. The determination of the automaton to be used is made on the basis of the first character of a scanned item. These three automatons are:

1. Letter -- The occurrence of a letter causes entry into a finite-state automaton, assembling a string which is either a reserved word or an identifier. This string assembly stops when a special character delimiter or a space has been read. the resulting character string, without the terminating special character, is hashed. The look-up table is entered via a table of pointers indexed by the hashed value. The identifier is matched with the identifier represented by the look-up table entry. The spelling

of the look-up table entry can be found via a pointer to the name and literal table. On a match, the look-up table index is returned as the token number. If there is no match, a hash code thread is followed to the next eligible entry and the comparison made again. When no match is made along the entire hash thread:

* The next available entry in the look-up table will be assigned to the new identifier;

* The proper spelling entered into the name and literal table;

* The look-up table entry attached to the hash code thread;

* The new look-up table index is returned as the token number.

The successful detection of a reserved word returns a unique pre-assigned token number for that reserved word.

When a non-reserved identifier is found and it is followed by a single quote (from the look-ahead), an 'Attribute Expected' flag is set to signal the Lexical Scanner that the next identifier should be checked for being a correct attribute. Then, when the 'Attribute Expected' is seen to be set and a letter encountered, the identifier is checked against a special table to detect the name of the attribute desired. This table must be kept separate of the reserved word/identifier look-up table because the attribute names are not reserved and must not be accidently considered as such. When the 'Attribute Expected' flag is set and anything other than a correct attribute is detected, an error token is returned. A detected attribute returns the identity of the attribute, not a pointer to any outside table. The correct spellings of the Predefined Language Attributes are found in Appendix A of the [DoD80].

2. Digit -- When a digit is encountered as the first member of a lexical unit, a finite-state automaton assembles and identifies the nature of the numeral. The automaton will identify all of the following:

   * Real decimal literal with or without an exponent;

   * Integer decimal literal;

   * Based number. The literal representation of the number is saved in the name and literal table without generating duplications of the same literal. The token for a numeric literal contains a pointer to the name and literal table entry. Illegal constructs cause an error token to be returned.

Any problem of detecting a real decimal literal which starts with a decimal point, e.g., '.123', is alleviated by Ada Language because the syntax defines a real literal as having a leading integer numeral, '0.123'.

When a "period" follows an integer, the decision as to whether or not the "period" is the decimal point of a real number or part of the ".." of a range constraint, depends upon the next character to be scanned. Therefore, the lexical scanner looks ahead two characters after the integer has been scanned.

3. Punctuation -- A special character appearing at the start of a lexical unit causes the recognition of that character and returns its identity. This serves to identify simple punctuation and operator symbols. The punctuation may be delimiters (cf. [DoD80B], Section 2 2) or compound symbols. Some look ahead is required in the case of the hyphen, '-', for the proper recognition of the minus sign and comments, the asterisk, '*', for the recognition of multiplication and exponentiation symbols, and the other compound symbols.

   The scanner recognizes and ignores comments. This is done by finding the string "--" and skipping the rest of the source record.

   Character literals and character strings are fully detected and placed into the name and literal table. The returned token contains a pointer to the entry in the name and literal table.

   This automaton may be entered when a single quote which is prefixing an attribute is encountered. The problem arises when distinguishing bewteen a character literal and attribute. The Ada Implementor's Guide offers a solution [SOFT80B] with the detection of a "apostrophe, character, apostrophe" string and a check to the last lexical element for a reserved or non-reserved word identifier.

3.2.1.4 Syntax and Semantic Analysis

The processing performed by the analyzer is depicted in Figure 3-7.

Syntax analysis is performed by a recursive descent parser. A practical grammar for Ada was derived from the concrete syntax given in the language reference manual [DoD80B] (cf. Appendix C). Conceptually, each nonterminal in the revised grammar is represented by a recursive routine that accepts any string derivable from that nonterminal, and only such strings. In actual practice, some of the trivial routines for nonterminals are merged with others to form composite routines. To carry out its task, the recursive routine performs lexical, syntactical, and semantical analysis, i.e., these three subtasks are interleaved in the logic of the routine, and builds the abstract syntax tree for the string.

The structure of each recursive routine is basically the same.

1. The control portion forms a tree from the tree(s) built by its recursive syntactical descendents. A tree contains both parse tree nodes and DIANA attributed tree nodes; DIANA nodes are parse tree nodes transformed as a result of semantic analysis. The resultant tree is passed back to the routine's caller.

2. The syntax portion obtains tokens from the lexical scanner. If the token is part of a legal syntactic construct, it is either processed by the routine or another routine is called to perform the processing; otherwise, an appropriate syntax error is generated. Eventually the construct to be recognized by the recursive routine will have been scanned and its syntax checked.

3. Then static semantic analysis is applied to the tree for the construct. This consists of applying the static semantics as given in the formal definition [DoD81]. This approach permits the correctness and completeness of the analysis to be demonstrated. This results in a tree with both DIANA and parse tree nodes. Any DEF_IDs occurring in the attributed tree are entered in the symbol table.

Thus, as recursive routines are left, a progressively more complex and complete tree is built until a fully developed DIANA tree exists for the target construct. Notice that certain aspects of Ada cannot be processed in a single tree traversal. For example, overload resolution requires the constructed expression tree to be traversed twice (cf. Appendix D).

The abstract syntax tree is a condensed tree representation of the derivation tree for the derivable string. It serves as the intermediate language between separate parts of the compiler and contains information needed for performing transformations on the string, e.g., optimization and code generation. This semantic information is distributed as attributes in the nodes of the AST and the entries for symbols in the symbol dictionary. The information is used in the performance of semantic analysis tasks, e.g., of type checking, control of the visibility of declared items, and resolution of overloaded operators (i.e., operator identification). The entries for symbols are actually IL nodes referenced from AST nodes. They also can reference AST nodes, e.g., unevaluated initialization values. The error recovery mechanism employed guarantees that the AST is syntactically correct (cf. Section 3.2.1.5.3).

After the compilation specification of the compilation unit is processed, the nature of the compilation unit is determined. If it is a library unit and this is the first time it is being compiled, an entry for it is made in the library file (cf. Appendix F). If it is a subunit which is a subprogram, the subprogram specification in the body is checked to see if it corre^ponds to that given in the body stub.

During the processing of the compilation unit, subunits defined by a body stub are registered in the library file, i.e., an entry is made for the subunit.

If compilation of the compilation unit is successful, the library file is updated to reflect the new compilation state by replacing it with the updated memory copy (cf. Appendix F).

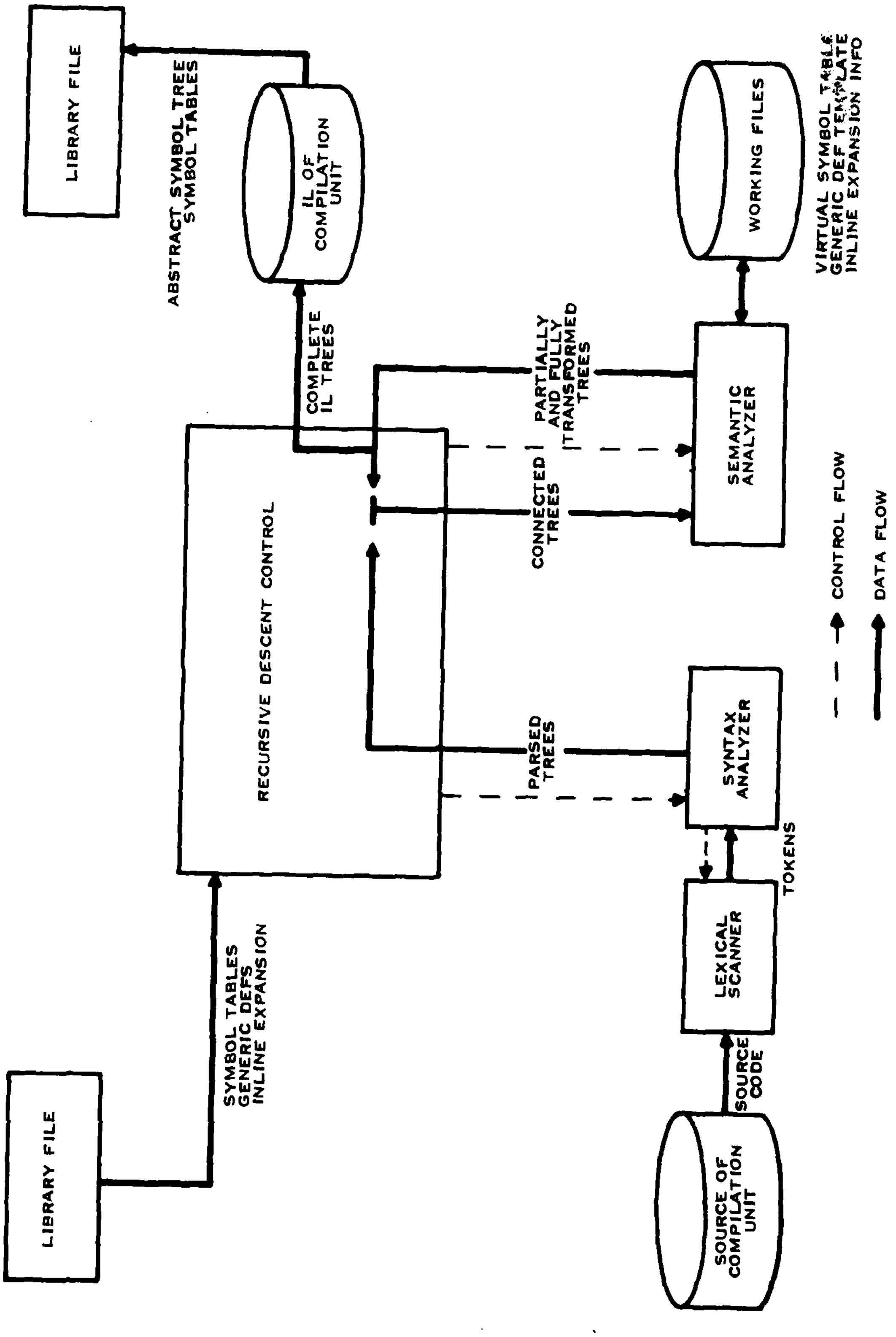


Figure 3-7 Organization of the Front End

3.2.1.5 Semantic Analysis Issues

3.2.1.5.1 Enforce Visibility Rules

The symbol table is a compiler data structure that associates symbols with their attributes. A symbol can be an identifier, an operator, a literal, predefined attributes and pragmas, and reserved words. The attributes depend on the symbol; typical attributes are type, name, initial value, and scope information. Values are assigned to the attributes as they become known. `There is a separate symbol table for each language construct that can contain a declarative part, i.e., blocks and program units (subprograms, packages, and tasks), which is saved as part of the construct's IL (cf. Appendix E, Section E.4.3). Symbol tables are assembled together to form the compilation context for a block or program unit and the entries are interconnected together in a manner effecting the semantics of the visiblity rules (i.e., scope, hiding, overloading, USE, WITH, RENAME, and the predefined environment). The resulting data strùcture, known as the symbol dictionary, is described in Appendix B, Section C.1. It contains the information needed by the different passes of the compiler.

Appendix B also attempts to demonstrate informally that the design of the symbol dictionary supports the enforcement of the visibility rules by stating the required processing. This processing consists of:

* creating the predefined environment (cf. Appendix D, Section C.2)
  - initializing the symbol dictionary, in particular, for the package STANDARD, the reserved words, and the predefined pragmas
* maintaining the compilation context (cf. Appendix D, Section C.3)
  - keeping the block structure
  - maintaining the scope of declared entities
  - processing packages (visible and private parts, private types)
  - processing a WITH and USE clause
* enforcing the visibility rules (cf. Appendix D, Section C.4)
  - direct visibility for a block, subprogram, task, and the visible part of a package (via a USE clause)
  - hiding
  - handling a loop parameter
  - handling selected components

- handling overloading for subprograms, enumeration literals, and aggregates

- operator identification

* handling renaming, both static and dynamic (cf. Appendix D, Section C.5)

* handling the associated inheritance of subprograms and operators for derived types (cf. Appendix D, Section C.6)

3.2.1.5.2 Static Expression Evaluation

Static expressions needing to be evaluated, e.g., to perform compile-time error detection or type determination, are calculated using a package that simulates the arithmetic operations of the target machine specified via the SYSTEM pragma. Another target machine dependent package is used to obtain values of language defined attributes that may appear in a static expression.

3.2.1.5.3 Error Handling

Error recovery is performed in the fashion described by Hartman [HAR77] and later elaborated upon by Pemberton [PEM80]. This is a relatively straightforward technique consisting of skipping symbols until an appropriate symbol occurs in the current context. When it is possible to determine the nature of the error, i.e., when a semicolon is expected, an error message is be emitted and parsing continues as if the expected symbol had occurred.

There are cases where it may be feasible to move this up a level. For example, when an arithmetic operator is expected, parsing can continue by emitting IL to indicate an unspecified (erroneous) operation.

Every effort shall be made to give the most meaningful error message possible. This sometimes entails being careful about specifics since as more specific details are put into an error message, the chances are also increased that this message may sometimes be wrong. A classic example of this is the message emitted by many FORTRAN compilers relating to function statements when the real problem is a missing dimension on an array.

The DIANA produced by the front end is syntactically correct so subsequent passes can ignore processing of syntactic errors. This may require deletion of some previously produced DIANA nodes.

3.2.1.5.4 Generics

There are five times during the compilation process that generics are processed: 1) when the generic declaration is met, 2) when the generic body is met, 3) when the generic instantiation is met, 4) during expansion of a generic instantiation, and 5) during the optimization of the generic instantiations of a given generic declaration. The generic declaration/body and instantiation are processed by the analyzer. The generic body is transformed into an IL template to be used during the expansion. Generic instantiation involves substituting the actual parameters for the generic formal parameters in a copy of the generic specification. Expansion of a generic instantiation results in replacing generic formal parameters in the IL template for the generic declaration with the actual parameters. Generic expansion is performed by the expander/optimizer pass (cf. Section 3.2.2.3.2.2). Further processing results in a customized expansion. Optimization of generics is an attempt to share code between different instantiations of a generic definition. Whether or not code can be shared is a function of both the target machine and the program. Details of the kinds of generic optimizations that can be performed and the generic optimizer is given in Section 3.2.1.5.4.3. Generic optimization is performed by the program binder. If no optimization is possible, the customized instantiation is used to build the object module.

3.2.1.5.4.1 Processing a Generic Declaration and Body

The definition of a generic subprogram/package consists of two parts: the generic subprogram/package specification and the body. These two parts do not necessarily have to be adjacent to one another in the source text nor need they be compiled together. Processing the generic declaration results in a symbol table for the generic unit; it contains the name of the generic unit and the generic formal parameters. The body of a generic subprogram or package is processed like a normal subprogram or package. Because of the way generics were, designed there should not be any unresolved type or overload resolutions to be performed, e.g., a user must supply as a generic formal parameter any subprograms needed to resolve overloading. Therefore, operator identification, subprogram overloading resolution, and type checking can be performed as the body is processed. (The matching rules make sure the actual parameter correctly matches the formal parameter.) The resultant AST is a template for the AST of corresponding program units obtained by generic instantiation. The IL for the generic unit (AST and symbol table) is stored in the IL file (cf. Appendix E, Section E.4.3) for the compilation unit. A list of all generic declarations and bodies within the compilation unit is recorded in the entry for the compilation unit in the library file (cf. Appendix F).

The top portion of Figure 3-8 depicts processing a generic declaration and body.

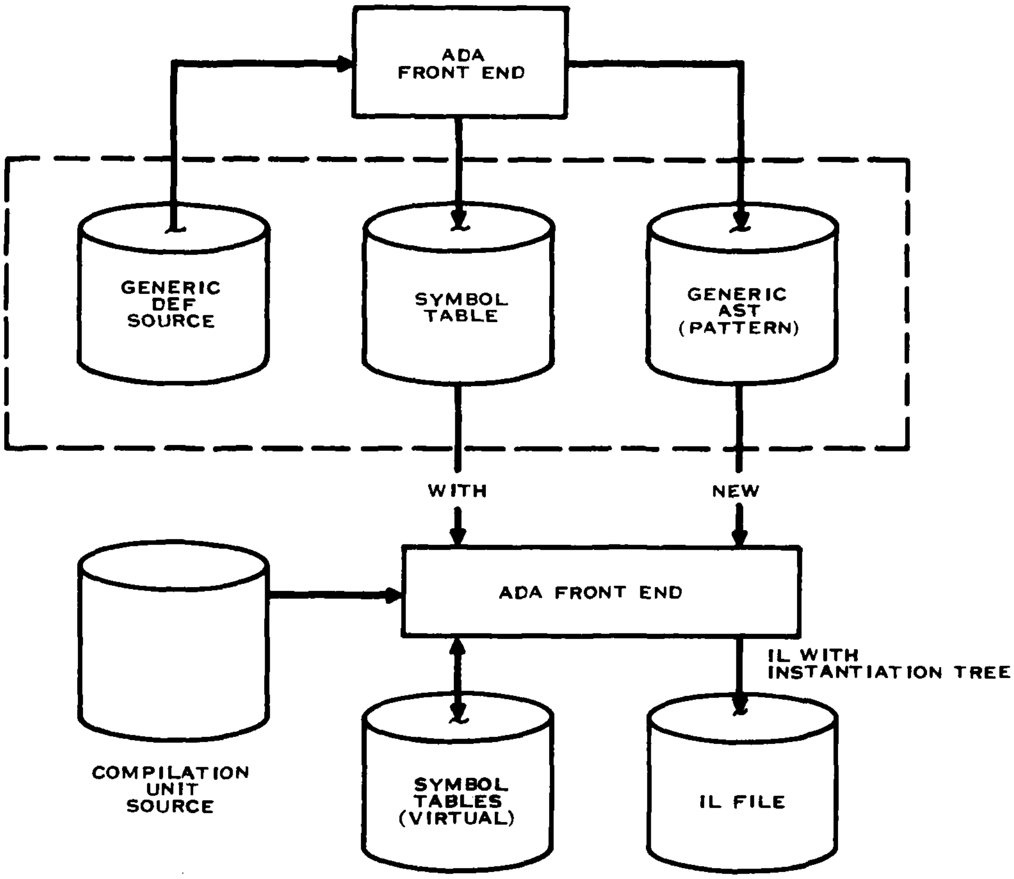


Figure 3-8 Generic Definition and Instantiation Data Flow

3.2.1.5.4.2 Processing a Generic Instantiation

In DIANA, an instantiation is not expanded, i.e., the generic declaration copied and actual parameters substituted in the copy for the generic formal parameters. This could not be done in general by the front end since the body of the generic unit may be compiled separately. Instead, an instantiation involves copying only the specification part of the generic unit and replacing in the copy every occurrence of a generic formal parameter with the corresponding actual one. The matching rules are applied between the actual parameters and the generic formal parameters.

The bottom portion of Figure 3-8 depicts processing a generic instantiation.

3.2.1.5.4.3 Optimization of Generic Instantiations

Given as input a reference to the IL for a generic unit and a list of all instantiations (this information is obtained from the library file), the generic optimizer consolidates identical instantiations into one and finds instances where code may be shared. With respect to sharing code, the kinds of optimizations that can be performed depends on the target machine. Typical examples are:

* for types having the same representation, any generic instances can be shared.

* for machine operations which are type independent, e.g., a comparison operator or data movement, or which decode an operand descriptor as in a tagged architecture, any generic instances involving only these operations can be shared.

* for type dependencies whose frequency and/or size relative to the total code is 'small', generic instances involving such a dependency are candidates for either procedure parameters and/or casing.

For those generic instantiations which can be shared, either one of the expansions is used or the IL is reformed for the new expansion (as would be the case if additional parameters were being passed or there was casing). If new IL is formed, it is automatically passed through the expander/optimizer and code generator. The object module for the new expansion is passed back to the program binder. For the case of a generic subprogram which has a formal subprogram parameter, if the instantiations can be shared and the actual parameters are not inline subprograms, the formal subprogram parameter is implemented as an implicit parameter.

Optimizing generics across a program may result in slow link times. If this proves to be the case, it shall be restricted to a compilation unit. Regardless of when it is applied, the processing required to perform generic optimization is the same.

Generic optimization is applied to the compilation units involved in the segment binding phase of the program binder, but not to partial or fully formed program segments. This allows such a partially linked object module to be shared and implemented in ROM.

3.2.1.5.5 The INLINE Pragma

The INLINE pragma denotes a property placed upon the names of the subprograms appearing within a declarative part of a program unit or block. An instance of the INLINE pragma in a declarative part, results in entities having that name, including overloaded subprograms declared in that declarative part, as having the inline property. When the user wishes to intermix the inline subprograms within an overloaded set, the user may have to resort to separate compilations of the inline and non-inline subprograms to insure proper intent. Subprograms which are compilation units (as is the case of subprograms which are library units or separately compiled subprogram bodies), not having an encompassing declarative part may have the pragma inline declaration as the first statement of the compilation unit.

Calls to a subprogram before it is known that the called subprogram is to be inline (e.g., the INLINE pragma appears with a separately compile body and not the corresponding specification or an overloaded subprogram is specified later in an INLINE pragma) results in a normal calling sequence code. When the body of an inline subprogram is met, all calling subprograms compiled before the body of the inline subprogram may have to be recompiled. A warning is issued by the compiler to this effect with a list of all calling subprograms. Separate recompilation of a call results in inline expansion. Expansion of inline subprograms is handled by the expander (cf. Section 3.2.2.3.2.1); the analyzer only builds a DIANA pragma node.

There are a number of issues related to recompilation. When the body of an inline subprogram is changed and recompiled, the caller is changed and must be recompiled. Also, if an inline subprogram is recompiled and no longer specified as inline, all callers must be recompiled. Further discussion of these issues can be found in the discussion on separate compilation in Section 3.2.1.2.

3.2.1.5.6 Representation Specifications

When the representation specification is met in a declarative part, a DIANA abstract syntax tree (AST) is built for it. A representation specification for a type must appear in the same declarative part containing the declaration of the type. There may be more than one representation specification, each of which specifies different aspects of the representation. To facilitate later processing, an implementation attribute, rep_s, for representing the representation specifications for a type is added to the DIANA DEF_ID node defining the type. The value of the attribute is a reference to the root node of the ASTs for the representation specifications. The static expressions in a representation specification are evaluated by the expander. The representaton specification is used by the code generator in the allocation of items.

3.2.1.6 Saving the IL

During the processing of a program unit (subprogram, package, task) and block, its abstract syntax tree and symbol table are constructed. The symbol table represents a description of the information contained in the DIANA AST (in DIANA each definable entity is represented by a defining occurrence, i.e., an AST node [GOO81, Section 1.1]). This dichotomy facilitates later processing which requires the symbol table, e.g., construction of the compilation context and source level debugging. The structure of the AST is determined during syntax analysis, i.e., nodes are created and appended to the tree. During static semantic, analysis the nodes are decorated with DIANA attributes. After each program unit and block is processed, its IL (AST and symbol table) is written out to the IL file. The structure of the IL file is described in detail in Appendix E, Section E.4.3. During the processing of embedded program units and blocks, the IL (AST and symbol table) for the enclosing unit is retained in memory. Once the IL is written out, the memory space is reclaimed. If the compilation unit is being recompiled, all IL files and object modules previously generated from it are discarded, i.e., the library file is updated to reflect the new compilation state of the unit.

3.2.1.7 Compilation Statistics

Generation of compilation statistics are controlled via the implementaton defined language pragma STATS, which can only appear before a library unit. It takes COMPILER and/or STATIC as arguments. The STATIC statistics pertain to static characteristics of the program and are partitioned into the following categories:

* Program structure characteristics: distribution of statement types, distribution of types, distribution of the number of program units and blocks, distribution of nested control constructs, distribution of the number and usage frequency of INLINE subprograms, distribution of the number and usage frequency of generics, distribution of the number of statements, etc.

* Declaration characteristics: distribution of initialized variables, occurrences of overlays, distribution of the number, types and kinds of parameters per procedure.

* Statement characteristics: distribution of forward and backward control transfer distances (in tokens), distribution of number of operators per statement type, distribution of operator type by statement type, distribution of number of operands per statement type, distribution of operand type by statement type, distribution of loop types, number of implicitly delcared loop control variables.

* Expression characteristics: distribution of expression length in operands and operators.

* Operator characteristics: distribution of operator types per compilation unit

* Operand access characteristics: distribution of constants by type, distribution of operand types, number of operands per compilation unit, distribution of the number of array indices, distribution of operand reference scope (local, global, parent, intermediate).

COMPILER statistics pertain to compiler performance and consist of:

* rate of processing.

* memory requirements

* disk accesses

* elapsed CPU time

* elapsed clock time

* number of instructions generated for each kind of DIANA node

Statistics generated by a compiler pass for a compilation unit are written to a file associated with the library file: one file for static statistics and another file for compilation statistics. The static and compilation statistics shall be accumulated and summarized by a separate AIE tool. This tool shall also be capable of extracting statistics generated by a particular compilation of a compilation unit.

3.2.1.8 Formation of the Statement Map

The Ada source level debugger needs a statement map in order to map machine code back to the source. The statement map is initially generated by the analyzer and updated by each pass of the compiler that performs optimizations. There is one statement map file associated with each compilation unit. This file contains a statement map for the compilation unit and each embedded program unit and block; as each is processed, its statement map is appended to the file. A statement map consists of the name of the program unit or block and for each statement the following:

* the line number of the statement in the source listing

* the source statement number

* the offset within the program unit or block of the first byte of the first instruction statement

* the offset to the first byte of the last instruction of the statement

* a list of references to symbol table entries of statement labels (if present).

The last two pieces of information are filled in by the code generator.

3.2.1.9 Outputs

The outputs of the analyzer and the default file pathnames are summarized in Table 3-2. Their generation is discussed in previous sections.

Table 3-2 Outputs of the Analyzer

```
IL file:
     library_file.library_unit{.subunit}.DIANA
Compiler Statistics (optional):
     library_file.LIST.SSTATS
     library_file.LIST.CSTATS
Statement Map:
     library_file.library_unit{.subunit}.ASMAP
```

The library file is updated to reflect the new compilation state of the compilation unit. In particular, the database names of the generated IL file, statement map file, and type representation specification file are recorded in the compilation unit's entry.

The pathnames, default or user specified, for the output files in Table 3-2 are also output.

3.2.2 The Expander/Optimizer: Introduction

The expander/optimizer is divided into two parts: an expansion-local machine-independent optimization subphase and a global machine-independent optimization subphase. The expansion subphase makes run-time representational choices about data types and control structures, and incorporates the choices into the DIANA abstract syntax tree. After the expansions are made, local machine-independent optimization is performed. The global optimization subphase performs global machine-independent optimizations; the nature and level of optimization is controlled by the pragmas OPTIMIZE and OPTLEVEL, respectively. The net result is a lower semantic level dialect of DIANA which is machine dependent. Applying global optimizations to this lower level dialect will be more effective than if optimization was applied to the dialect output by the analyzer.

The unit of processing is the body of a program unit (subprogram, package, or task) or a block. Program units and blocks within a compilation unit are processed in the order determined by their lexical level in the source text, i.e., from the outermost lexical level to the innermost and for program units and blocks at the same level, in the order of occurrence. The processing order is the reverse order in which the IL was written out (cf. Appendix E, Section E.4.3). Prior to the processing of each unit, its compilation context is created and its symbol table loaded.

Table 3-3 lists the machine-independent optimizations performed by the expander/optimizer. The local optimizations are performed by the expander, while the global optimizations are performed optionally by the global optimizer. Other optimizations, such as register allocation, variable overlaying, structure alignment, peephole, inline substitution, reduction of Boolean expressions, and algebraic simplification of subscription, are performed in the code generation pass. Generic optimization is performed by the program binder.

Table 3-3 Machine-Independent Optimizations

```
Local Optimizations
     Constant folding
     Constant propagation
     Common subexpression elimination

Global Optimizations
     Assertion propagation
     Common subexpression elimination
     Code motion
     Strength reduction
     Loop unrolling
     Loop fusion
     Dead code elimination
     Dead variable elimination
```

3.2.2.1 Inputs

Input to the expander/optimizer is the pathname of a control file (cf. Section 3.1.1.3). The control file contains the pathname(s) of IL files output by the analyzer to be processed. The dialect attribute of the IL file is used to verify that the correct DIANA dialect is input. The control file must also specify the pathname of the library file in which the compilation state of the compilation unit(s) is maintained. The optimization pragma OPTIMIZE and OPTLEVEL pragmas for a program unit or block may be specified in the source text or respecified in the control file.

3.2.2.2 Pre-processing

Prior to performing any expansions or optimizations, an information gathering subphase performs the following:

* The AST for each block and program unit within the compilation unit being processed is traversed and the control flow graph is constructed. Each node of the flow graph represents a basic block. A basic block is an ordered set of DIANA nodes with a single control path through them; entry may be only to the first node and exit may be only from the last node. The control flow graph is the basic data structure used by the flow analysis algorithms. Other information required by the optimization algorithms can be extracted from it, e.g., a list of immediate successors and immediate predecessors for each basic block; data flow relationships, e.g., definition, use, and live range for variables and temporaries; the loops and their entry block; and the immediate predominator for each basic block.

- * Static expressions not evaluated by the front end are evaluated. As in the front end, this evaluation is target machine dependent and uses a package that simulates the arithmetic operations of the target machine specified via the SYSTEM pragma.

- * Unreachable code is eliminated. Examples are sequences of statements in an if statement that will not be executed because of the static condition. Or, choices in a case statement may not be executed because of the static expression used to select the choice.

- * The static frequency of access for each declared object is determined. This information is used by storage allocator (cf. Section 3.2.3.2).

3.2.2.3 Expansion/Local Optimization

Representational decisions about data and control structures are made, after which these decisions are made explicit in the AST.

3.2.2.3.1 Representational Decisions

Based on the Ada virtual machine (cf. Ada Software Environment CPDS), the packing algorithm chosen for the target machine, and representation specifications, a target machine dependent representation (i.e., size and internal structure) is chosen for each data type. The size information is used to determine the size of the actual parameter list for each subprogram call node. These choices are encoded in attributes of AST nodes.

Representational choices for a number of control structures are also made. Based on the size of an IN formal parameter, it is decided whether the actual parameter is to be passed by copy or by reference. The decision is encoded in the defining occurrence of the formal parameter. Based on the number of choices and their sparseness, the implementation of each case statement is chosen, i.e., as nested if statements or as a transfer vector. The decision is encoded in the case node.

3.2.2.3.2 Expansion of the Abstract Syntax Tree

Based on the preceeding decision making, a number of high-level language dependent tree transformations are performed on the AST.

3.2.2.3.2.1 Inline Subprogram Expansion

The rules for determining whether an inline subprogram can be expanded inline are:

1. If the body of the inline subprogram has not been compiled, the inline is ignored and the call is elaborated with a normal calling sequence.

2. If the body has been compiled and if the caller is not inline,

the called inline subprogram is expanded.

3. If the body has been compiled and if the caller is inline,

   a. Construct a call graph starting with the caller as the root.

   b. Find all recursive cycles involving the caller and the inline subprogram.

   c. If there exists one recursive cycle in which all subprograms are inline, the INLINE pragma of the called subprogram is ignored and the call is elaborated with a normal calling sequence.

   d. If no recursive cycles exist with all inline members, expand the called subprogram normally.

Inline expansion takes place as an integration of the DIANA of the callee with the DIANA of the caller at the point of call. The IL (AST and symbol table) of the caller is modified so the meaning of the subprogram [DoD80B Section 6.3] and the semantics of the parameters are preserved. Inline expansion proceeds as follows:

1. A copy of the symbol table and DIANA abstract syntax tree of the called subprogram is made. The subprogram call node is replaced by the DIANA AST.

2. References to IN OUT and OUT parameters within the expanded subprogram are changed to the actual parameter's symbol table node.

3. References to IN parameters within the expanded subprogram reference a compiler generated temporary variable with the type of the actual parameter. AST nodes are inserted prior to the call node which assign a copy of the actual parameter value to this compiler temporary variable. A symbol table node defining the compiler temporary variable is placed in the symbol table of the caller. Literal values passed as IN parameters are directly substituted into the syntax nodes of the called subprogram, allowing for possible optimization in a later pass of the compiler.

4. A RETURN value is placed in a compiler generated temporary variable assigned to the caller. This temporary variables is normally generated for a return value and handled by the caller when program control returns. AST nodes are inserted after the call code which assign the return value to this compiler temporary variable.

5. Symbol table nodes denoting local variables of the inline subprogram are logically merged with the symbol table of the

> caller. This is achieved by recording in the entry for the compilation unit containing the caller a reference to the callee's symbol table. When the expanded AST is processed, the symbol table loaded for the caller consists of its symbol table generated by the analyzer plus the symbol tables of all expanded inline subprograms. Duplication of identifiers is not a problem because all name conflicts have been resolved and the IL contains a unique reference to the proper identifier, i.e., node_name's between different compilation units are unique (cf. Appendix E, Section E.4.1). The size of the activation record of the calling subprogram is increased to handle the new local symbols.

A function which is used as a parameter to a subprogram would normally be evaluated prior to the call to the subprogram and the parameter value set from a temporary variable. An inline function would be likewise expanded prior to the subprogram invocation and the parameter value assigned from the resulting temporary result.

3.2.2.3.2.2 Expansion of a Generic Instantiation

To perform the expansion, the generic body must have been compiled. If the compilation has not been performed, an error message is issued to this effect with the name of the generic unit. If the generic body has been compiled, expansion of a generic instantiation involves replacing generic formal parameters with the actual parameters in a copy of the generic body's AST (cf. Figure 3-9). This is a simple substitution process since all type and overload resolution were performed by the analyzer. The substitution involves changing references to formal symbol table nodes (for types, IN OUT variables, and subprograms) to references to actual parameter symbol table nodes. For an IN variable, the AST for the expression representing the value replaces the formal parameter.

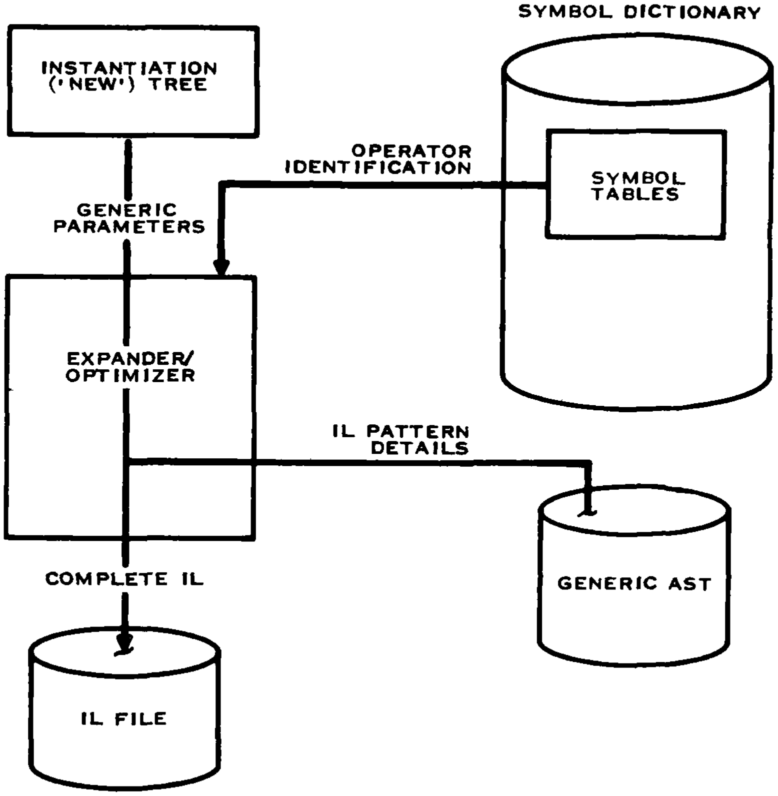


Figure 3-9 Generic Expansion Data Flow

A symbol table of all symbols defined in the expansion is constructed by copying nodes from the symbol tables of the generic body and the units defining the actual parameters. The symbol table and IL for the expansion are saved on an IL file (cf. Appendix E, Section E.4.3).

3.2.2.3.2.3 Other Tree Transformations

The following transformations are made on the abstract syntax tree:

* The passing of actual parameters in a subprogram call are made into explicit assignments to the actual parameter list, a compiler gnerated temporary. (The actual parameter list is allocated in the

stack frame of the caller (cf. Ada Software Environment CDPS)). Ths assignment is based on whether the formal parameter is passed by copy or reference. This compiler temporary is subject to the same optimizations as a user defined object. In particular, since its lifetime is known, actual parameter lists may be shared. Also, since the actual parameter list is read only, its values are invariant after return from a subprogram call. Code to initialize actuals that are constant may be moved out of a loop.

- Function returns are expanded into an assignment to a compiler generated temporary. During storage allocation in the code generation pass, this compiler temporary will be allocated to a dedicated register that either contains the functional value or a pointer to the functional value. The compiler temporary is subject to the same optimizations as a user defined object.

- Address arithmetic for array subscripting, up-level addressing, referencing accessed objects, and referencing a selected component are made explicit. The required computation is based on the representation choice made for the data type. Exposing the address calculations permit them to be optimized, e.g., for common subexpressions to be detected. The level of expansion for array subscripting depends on the architectural support of the target machine.

- Check nodes not designated as suppressed by the *SUPPRESS* pragma are expanded into explicit tests, e.g., checks on range error in array subscripting and scalar assignment are generated. Unnecessary check nodes are removed by the assertion propagation algorithm during global machine-independent optimization.

- Address and value contexts are explicitly distinquished. This information is useful during code generation.

- Certain control constructs are expanded into more primitive operations. For example, case nodes are replaced by an equivalent subtree of nested if statements or marked as using a transfer vector in effecting a choice.

- DIANA nodes for aggregates with static components are replaced by a node defining the constant. This permits the aggregate to be manipulated as a block instead of component by component.

3.2.2.3.3 Local Optimization

After all the expansions are made to the abstract syntax tree, local machine-independent optimizations are performed because of their high payoff. These optimizations, applied to a basic block, consist of constant propagation and folding, and common subexpression elimination.

3.2.2.4 Global Optimization

After all expansions and local machine-independent optimizations have been performed, the global machine-independent optimizations are performed (cf. Table 3-3). Assertion propagation is always performed. It is required in Ada for the elision of constraint checks and overflow checks; otherwise, code size and execution time will be intolerable. Whether or not the remaining global optimizations are performed is controlled by the presence/absence of the OPTIMIZE pragma. If the pragma appears in a declarative part, then the block or body enclosing the declarative part is optimized to the level specified by the OPTLEVEL pragma. If the OPTIMIZE pragma is present, but the OPTLEVEL pragma is missing, then all global optimizations are performed. If the OPTIMIZE pragma is absent, but the OPTLEVEL pragma is present, the OPTLEVEL pragma is ignored. OPTLEVEL takes an integer as an argument that specifies the level of optimization. The global optimizations are ordered, the ordering weighted by their potential payoff and execution time. The optimizations are grouped into classes, each class defining an optimization level. The integer argument of OPTLEVEL specifies that all optimizations up to an including those in the specified optimization level are to be performed.

The optimizer does not actually effect common subexpression elimination or code motion by making the necessary tree transformations; it only indicates which ones are feasible. The pre-processing subphase of the code generation pass chooses which of the feasible optimizations are, in fact, desirable.

The optimization algorithms employed shall be based on those developed by Texas Instruments, those used in the Bliss_11 compiler [WUL75] and the more recent versions used in the PQCC project [LEV75]; the work on assertion propagation [WEL78] shall be used for the assertion propagation algorithm. Texas Instruments has implemented machine-independent optimizers for TI Pascal, DX10 Rifle (a subset of TI Pascal used for systems programming), and Fortran for its dataflow architecture. Texas Instruments is currently under contract to the U.S. Army to design an Ada compiler for the dataflow architecture (Contract No. DAAK20-80-C-0276), and shall utilize its optimization algorithms. Also, Texas Instruments shall draw on newly published results, especially theses related to the PQCC project and specifications of the PQC phases, to improve the algorithms.

3.2.2.5 Saving the Expanded/Optimized IL

As each program unit and block of the compilation unit are processed, the expanded AST is written to a new IL file. With the exception of generics, the symbol table is not written out since it is the same as that generated by the analyzer. The structure of the IL file is described in detail in Appendix E, Section E.4.3.

3.2.2.6 Outputs

The optimized AST and flow graphs for the compilation unit and all embedded program units and blocks are written to a new IL file. The symbol table is not rewritten out to this file since it is unchanged; any pertinent code generation information is recorded as attributes in the AST. The pathname of this new IL file is:

```
library_file.library_unit{.subunit}.OIL
```

The statement map for the compilation unit output by the analyzer is updated when optimizations are performed. The database name of the statement map file to be updated is obtained from the entry for the compilation unit in the library file. The updated statement map is then written to a new file; the original statement map is retained since it may be needed in other derivations. The pathname of the updated statement map file is:

```
library_file.library_unit{.subunit}.OSMAP
```

The library file is updated to reflect the new compilation state of the compilation unit. In particular, the database names of the optimized IL file and updated statement map file are recorded in the compilation unit's entry; the input IL file is not discarded.

The pathnames, default or user specified, for the output files in Table 3-3 are also output.

3.2.3 The Code Generator: Introduction

To generate high quality code, an AST node must be examined in a context larger than a single node and the best code sequence for the context selected. Processing to generate code is divided into four phases: pre-processing, storage allocation, code generation, and post-processing. The pass is concluded with an object module formation phase.

The pre-processing phase performs various algebraic transformations and gathers machine dependent information, used in the later phases, defining the characteristics of the best code to generate.

The allocation phase assigns program declared objects and compiler generated objects to target machine registers and storage locations.

The code generation phase is table driven [CAT79, GRA80]. It generates code by matching subtree patterns against the context of the AST.

The post-processing phase performs final machine dependent optimizations on the object code prior to construction of the object module.

The last phase forms an object module in the format required by the program binder. Also, information required by the Ada source level debugger is generated.

The unit of processing is the body of a program unit (subprogram, package, or task) or a block. Program units and blocks within a compilation unit are processed in the order determined by their lexical level in the source text, i.e., from the outermost lexical level to the innermost and for program units and blocks at the same level, in the order of occurrence. The processing order is the reverse of the order in which the IL was written out (cf. Appendix E, Section E.4.3). This processing order guarantees objects referenced by embedded units have been allocated. Prior to a unit being processed, its compilation context is created and its symbol table loaded.

3.2.3.1 Inputs

Input to the code generator is the pathname of a control file (cf. Section 3.1.1.3). The control file contains the pathname(s) of IL files output by the expander/optimizer that are to be processed. The control file must also specify the pathname of the library file in which the compilation state of the compilation unit(s) is maintained.

3.2.3.2 Pre-processing

Prior to the allocation of storage and code generation, information required by these later phases is gathered and recorded in attributes of AST nodes, and source-to-source transformations are applied to the AST.

The pre-processor subphases are based on the DELAY phase of the Bliss-11 compiler [WUL75]. The contribution of each subphase to the production of optimal code is small; however, taken together the contribution can be substantial. The subphases are:

* Context determination: This subphase determines the way in which the result of an operation is used. Use of a Boolean is determined, i.e., whether to realize a Boolean result or utilize a conditional change in the flow or control. Also, arithmetic and address contexts are distinquished in order to utilize indexing or address instructions.

* Determination of desirable feasible optimizations: Common subexpressions and code motions involving an expression specified by the machine independent optimizer are examined. It is decided whether or not the optimization is worthwhile. For example, it may be more profitable to recompute a common subexpression at each use then to manipulate a compiler generated temporary or tie up a register over a region of code.

* Unary complement operator propagation: Unary complement operators are propagated to higher tree nodes until they are subsumed in other operations or they can be moved no further. Transformations may be performed on an expression that take advantage of target machine instructions (e.g., instructions which perform peculiar combinations of binary operators and unary complement operators).

* Evaluation order and targeting: Targeting makes use of the commutativity of arithmetic operators to avoid loads and stores. The algorithm employed to determine the optimal evaluation order for arithmetic expressions in the AST depends on the presence of

common subexpressions. During the processing, the number of registers necessary to evaluate an AST node is recorded in an attribute for the node. This information is used by the storage allocator.

The algorithms employed in each subphase can be found in [WUL75].

A final subphase performs target machine independent tree transformations [LEV75]. This subphase is driven by a table of transformation patterns: a subtree transformation skeleton describing the conditions that must be met in order for the transformation to be applicable and a subtree skeleton that describes the result of the transformation. Typically, algebraic transformations are performed.

3.2.3.3 Storage Allocation

The register allocator is machine-independent [SIT79A]. It accepts as input a set of objects that can be potentially assigned tc registers, the live range (obtained from flow analysis) and static frequency of use for each object, and a description of the storage hierarchy of the target machine. Using this information, the objects are mapped onto the storage hierarchy such that the most frequently used objects are allocated to high-speed access storage, and such that two objects with disjoint live ranges share the same storage location.

Prior to register allocation, all declared objects that must be allocated in the stack frame are marked. This selection is based upon the default/specified representation of the object's type, e.g., the length of a variable or alignment of a record. The actual stack frame displacements are assigned after register allocation. The set of objects not marked constitute the objects that can be potentially assigned to a register. The stack frame pointers of the most frequently accessed global objects are added to the set.

For target machines, like the IBM 370, which have base register addressing, the number of required base registers is determined and allocated prior to the execution of the register allocation algorithm.

For code space, the number of required base registers is based upon an estimate of the upper bound of the code size. This upper bound is the product of the number of DIANA nodes and the average number of instructions generated per node. (A more refined estimate is the sum of the product of the number of nodes of a particular kind and the average number of instructions generated for the node kind. The average number of instructions generated for DIANA nodes is obtained from compiler statistics (cf. Section 3.2.1.7)). If the size estimate is close to the range of the base register, only one base register is allocated. During code generation, if a second base register is required, the code generation pass is restarted with a second base register allocated. (This avoids unnecessarily allocating a base register and thereby generating less than optimum code.) If more than one base register is required, the flow graph is examined for execution locality, i.e., the absence of spagetti logic. If present, code is injected at appropriate points to load the shared base register with the correct base.

For data space, the size of the stack frame is computed (dynamic arrays present no problem since they are always accessed indirectly). The size information was determined in the expander/optimizer pass when the representation of each type was chosen. If more than one base register is required, the nature of the objects in the frame is examined. If a large object causes smaller objects to be allocated beyond the range of the base register, then the large object is placed at the end of the stack frame. Its original placement in the stack frame is replaced with a pointer; generated code accesses the object indirectly.

The number of available registers to which objects can be allocated is reduced by the maximum number of registers required to hold intermediate results. This number is obtained by calculating the maximum number of registers required for each control path in the flow graph. The register requirements for a node was determined in the pre-processing phase.

The allocation order of objects is based on frequency of use. Loop variables are weighted more heavily while large objects are weighted negatively. Using the allocation order, objects are mapped onto the top level of the hierarchy (i.e., registers), starting with the lowest offset. When the storage locations are exhausted, allocation is from the next level (i.e., stack frame), starting with the lowest offset. Thus, the most frequently referenced local objects are allocated to registers. Also, the stack frame pointers of the most frequently referenced global objects are allocated to registers. The most frequently referenced objects not assigned to registers are assigned to the start of the stack frame. The bias towards low offsets facilitates cache locality, the use of block move instructions (to initialize objects), and the use of instructions with short addressing modes.

The objects marked as having to be allocated in the stack frame are assigned stack displacements based on the type representation choices made in the expander/optimizer pass. The pointer for an access variable is assigned a stack displacement and the heap packet displacements within an accessed object are assigned. The storage assignments are recorded in attributes of AST nodes for later use by the code generation pass.

A typical storage hierarchy description is described by Sites [SIT79A]. The storage hierarchy is divided into a number of storage levels: even registers, odd registers, floating point registers, dedicated registers, index registers, general purpose registers, stack memory, and other memory. The storage hierarchy description (for a particular target machine) contains a description of each level. A level descriptor includes the number of available storage units, alignment, and permissible data types.

3.2.3.4 Code Generation

The code to be generated is based on the Ada run-time model (cf. Ada Software Environment CPDS). The exact code to be generated is not presented. Instead, the schema of how code is to be generated is presented, and for a few cases, the nature of the code to be generated is described.

3.2.3.4.1 Code Generation Schema

The code generation schema, based on the work of Cattell [CAT78], uses the information gathered in previous phases: register allocation, access mode determination, evaluation order, algebraic transformations, and so on. Their decisions are all represented in the DIANA representation of the unit by either explicit transformations or by decorating the tree with relevant information. Note, ancestor compilation units of a compilation unit must have been processed by the code generation pass so code can be generated to reference global objects.

The code generation phase is driven by a database of code generation templates. Each template is a pattern-action pair (abstract syntax subtrees and a code sequence to be generated). The code generation algorithm is described in detail in [WUL80, LEV79, LEV80]. Briefly, it consists first of traversing the AST and finding for each node all patterns that match. The best code sequence is selected and code is generated backwards. This has the advantage that a larger context is seen when traversing the tree backwards (top-down, right-to-left traversal) so that the best maximal pattern is applied. Efficient and comprehensive techniques exists to handle the enormous case analysis involved in the matching process [CAT78].

The pattern-action pairs are constructed by hand; they can be generated automatically using a description of the target machine as input when the code generator-generator technology becomes available.

To ensure code is generated, there is a pattern-action pair for each DIANA operation and every possible data move. The dynamic semantics portion of the formal definition and the underlying run-time model (i.e., the Ada abstract machine) is used to determine the target machine code to be generated for a DIANA AST node(s). There may be many code sequences for a pattern. The choices are based on the location or addressing mode of the operands, instruction set features, special case hardware, fast machine features, and the use of alternative operations. The patterns are sorted so special case patterns are examined before general case patterns.

Encoded with each instruction to be generated is its machine format. The generated code is represented as a doubly-linked list; nodes are linked together in the order the instructions will appear in the object module. This intermediate form is based on the data structure used in the FINAL phase of the Bliss-11 compiler [WUL75].

3.2.3.4.2 Processing Representation Specifications

A length representation specification of collection size results in a call to the run-time subprogram INITIALIZE_SUBHEAP (cf. Ada Software Environment CPDS) with one of its input parameters being the value of the expression representing the number of storage units to be reserved for the collection. This subprogram returns a subheap descriptor subsequentially used by the storage manager during the allocation of objects of the collecton.

An address representaton specification for an object results in code being generated which uses the address directly (addresses normally are displacements off a base or a register). An address specification for a subprogram, package, or task results in the generation of a linkage with an object mode tag indicating where the subprogram, package, or task is to be loaded. An address specification for an entry results in a call to the run-time routine CONNECT_INTERRUPT (cf. Ada Software Environment CPDS) with the address of the interrupt and the entry as input parameters.

3.2.3.4.3 Processing Language Attributes

Code generated to obtain the value of a language attribute involves either a compile-time constant (e.g., SIZE), access to a compiler known run-time data structure (e.g., attributes of a dynamic array), or a function call (e.g., COUNT, IMAGE, VALUE). For COUNT, the call is to a run-time routine whose input parameter is the address of the task control block (cf. Ada Software Environment CPDS). For IMAGE and VALUE, the code generated depends upon the type. For certain types, such as integer and real, the call is to an overloaded function whose input parameter is the parameter of the attribute, i.e., value and string respectively. For other types, such as enumeration, the call is to a function whose input arguments are a type descriptor and the parameter of the attribute. So that storage for the type descriptor is only allocated when an attribute requiring the descriptor is encountered, the function is generic. Its formal generic parameter is an IN mode object of the type descriptor's type. The call causes an instantiation of the generic function. The instantiation has the type descriptor literal bound into it. Duplicate instantiations for the same attribute and type shall be consolidated into one per type by the generic optimizer. To share code between different instances of the same type class, the run-time function calls an overloaded function whose input parameters are a type descriptor and the parameter of the attribute.

3.2.3.4.4 Processing Global References

Objects global to a compilation unit may have been allocated to a register by the register allocator. Addressability of the object is achieved by the fact a caller's registers are saved in its stack frame when a subprogram is called (cf. Ada Software Environment CPDS). The code generator is aware of the placement of the registers in the stack frame and generates code to access the appropriate saved register using the proper displacement from the base of the stack frame.

3.2.3.5 Post-Processing (Final Optimizations)

A final optimization phase performs machine dependent optimizations on the object code before the object module is constructed. The code is subjected to three classes of optimizations: branch and label optimization, peephole optimization, and cross-jumping [WUL79]. Application of these optimizations result in redundant store/load elimination, constant folding, dead code removal, a change in the sense of conditional branch instructions, the transformation of a code sequence into a more optimizal code sequence, etc. The optimizations are not performed in any order, but are inter-mixed and applied in multiple passes over the object code until no further optimizations can be performed.

The branch and label optimization and peephole optimization are executed in an alternating fashion. The branch optimizer processes the generated code in the forward direction. This permits the detection and removal of dead code, multiple labels, and branches to the next instruction, etc. The peephole optimizer processes the generated code in the backward direction [LAM80]. It has a sliding window through which small code sequences are examined and optimized. Cross-jumping optimization is applied as the first optimization pass and when ever the other two optimizers fail to perform an optimization. Any transformation of code by an optimization pass may result in code subject to another optimization. For instance, cross-jumping optimization can create branches which result in dead code. A later pass of the branch optimizer removes the dead code. As long as any optimizer performs an optimization, the generated code is reprocessed. Optimization is complete when all optimizers fail to perform an optimization.

The generated code is represented in doubly-linked nodes (cf. Figure 3-10). This structure allows the free traversal of the chain and the moving and/or deleting of code.

UP LINK
DOWN LINK
NODE KIND (LABEL, CODE)
OPCODE
INSTRUCTION LENGTH
NUMBER OF OPERANDS
OPERAND 1 - ACTUAL
ADDRESS MODE 1
SYMBOL NODE 1
OPERAND 2 - ACTUAL
ADDRESS MODE 2
SYMBOL NODE 2

Figure 3-10 Representation of a Code Node

3.2.3.5.1 Branch and Label Optimization

The following branch and label optimizations are applied to the object code:

1. Multiple Label Removal -- Multiple label nodes in the code chain which mark the location of an instruction node can be reduced to a

single label. To set references to the eliminated labels to the surviving label, the eliminated label identifiers are saved in a table. When references to these labels are encountered, the reference is set to the remaining label.

2. Unused Label Removal -- Label nodes in the code chain which are not referenced in the code are removed.

3. Branch Chaining -- Branch instructions whose target is a labeled branch instruction and long chains of these branches are optimized by setting the target label to the label of the last branch in the chain. Further optimization removes unused labels.

4. Dead Code Removal -- The nodes in the generated code chain which reside between an unconditional branch and a label node are unreachable and, therefore, removed from the code chain. This optimization is most likely to be used to remove dead code created by earlier optimizations.

5. Conditional Branch Optimization -- The instruction following a conditional branch is checked for the occurrence of an unconditional branch. If found, the sense of the conditional is reversed and the unconditional branch deleted. However, if both branches are to the same label, the conditional branch is removed.

   The next node in the object code chain is checked for a label node with the same identifier as the target of the conditional branch. If found, the conditional branch is removed.

3.2.3.5.2 Peephole Optimization

Peephole optimization consists of scrutinizing a few instructions at a time and transforming a recognized sequence into a more optimal sequence. The sequences generally result when optimum code generated for disjoint AST nodes is brought together, resulting in redundancies and/or an awkward code sequence.

The peephole algorithms are machine-relative, i.e., they are driven by a target machine dependent pattern table [DAV80] which is encoded in a machine independent format [DAV80]. A pattern is a boolean predicate involving, for example, opcode and addressing modes. The peephole optimizer scans the object code, viewing the code through a small window. The code appearing within the window is analyzed by comparing it to a set of patterns. Information is extracted from the instruction sequence being examined and a hash lookup used to reference the list of applicable patterns. The applicable patterns are ordered according to their payoff. The list is searched for the first boolean predicate that is true. For the pattern matched, pertinent data is extracted from the original code, as dictated by the pattern, and substituted into the form of the optimized code. This new code replaces the original code sequence.

The benefits derived from a pattern driven peephole optimizer have been documented in terms of up to 40% savings in final output object code [LAM80,DAV80,WUL75]; the percentage depends on the quality of the originally generated code. The following optimizations are performed by the peephole optimizer:

1. Statement Combining -- This part of the peephole optimizer represents the most labor intensive and machine dependent portion of the optimizer. In statement combining, the code chains are searched for patterns in adjacent instructions. Matching patterns in the code to pre-established sequences allows the optimizer to substitute single code instructions for two and three instruction sequences.

2. Code Substitution -- This optimization is a sub-set of statement combining. In code substitution, single instructions are substituted for longer and slower single instructions. For instance, some machines allow the use of long and short branches, short branches using only a single word to hold the opcode and range (1 to 127 words) of the branch. As the code is reduced by optimizations, longer branch instructions may come into the range of the short branch and be eligible for the shorter representation. Algorithms for long vs. short addressing determination shall be used for design guidance [ROB79].

   Another code substitution situation presents itself in the occurrence of literal operands in 'add', 'subtract', 'or', 'and', et.al., instructions. In many cases, the instructions may be changed to 'clear', 'set', 'increment', or 'decrement' instructions. Elimination of some code sequences may be possible through constant folding.

   The manipulation of constants by the optimizer helps eliminate unnecessary exceptions (divide by zero, etc.) and range checking. These checks would normally be made against variables at run-time, but in the case of constant values, compile time checks are just as valid and more efficient.

   A further optimization might be possible in the case of the 'increment' instruction. If the target machine has an auto-increment addressing mode, the increment may be combined with an earlier instruction.

3. Test and Compare -- Specific instructions which test a condition and set a condition code may be eliminated if the previous instruction, a move or arithmetic instruction, already has set the condition.

4. Ineffective Code -- An optimization is possible when code causes the storing of a value into memory, modifies that data and stores another value into the same memory location. If the modifications of the data are not used (moved to another location or loaded into a register, for instance), those modifications are removed as

ineffective code [WUL75].

5. Redundant Loads -- Instructions which cause the loading of registers with data stored in the previous instruction may be removed from the object code, if an active label is not attached to the load instruction.

3.2.3.5.3 Cross-Jumping

When two code sequences merge via an unconditional branch to a label node, it may be possible to remove redundant code if the code which leads to the merger is the same in each path. Cross-jumping is accomplished by backing up the code sequences preceding the two branches that merge at a label node and comparing them until a match fails. The first instruction of one sequence is replaced by a branch to the corresponding code node in the other sequence. This method can generate dead code which is removed later by the branch and label optimizer.

3.2.3.6 Formation of Object Module

Using the double linked list of instructions and internal tables constructed by the code generator describing constants and external references for each program unit, an object module for each program unit is constructed. Each object module has two control sections, viz., a code section and a constant section. For a subprogram or package body, any code produced during the compilation of its specification is concantenated with that generated for the body (cf. [DoD80B] Section 10.4.3). The format of the object module is as required by the program binder (cf. Program Binder CPDS). Literals are characterized for the binder as to whether they are local or global. As each object module is built, it is appended to a file resulting in one object file for the compilation unit.

3.2.3.7 Formation of Symbol Information for the Source Level Debugger

Information required by the source level debugger about each symbol defined in a program unit or block is recorded in three different places: the symbol table (generated by the analyzer), and the symbol map and type map. The latter two maps represent machine dependent attributes generated by the code generator. This dicatomy eliminates the need to save an updated symbol table; the source level debugger can ascertain all the information it needs from the symbol table, symbol map, and type map. From the node_name of the symbol's name in the symbol table, the node_name for the definition of the identifier can be obtained; it contains the node_name for the definition of the identifer's type. The symbol table entry for the type gives such informaton as references to subtypes, structure of a record or array, etc. The type's node_name can be used to look up the type's representation specification in the type map.

The symbol map and type map are also necessary for obtaining the value of langauge attributes. The entity following a prime is either an object, type, or entry. The symbol map is used to obtain attribute values pertaining to objects, and the type map for those pertaining to types.

There are circumstances where both tables are needed. For example, for a constrained array, in order to get to the array descriptor that contains the attribute value it is necessary to go through the symbol map (to obtain a type map reference) and then through the type map (to obtain the reference to the array descriptor).

3.2.3.7.1 The Symbol Map

The symbol map provides the map between symbol addresses and the symbol names. There is one symbol map file associated with each compilation unit. This file contains a symbol map for the compilation unit and each embedded program unit or block; as each is processed its symbol map is appended to the file. A symbol map consists of:

- the name of the program unit or block

and for each uniquely defined symbol:

- the node_name of the entry in the symbol table for the symbol's name (cf. Appendix E, Section E.4.1)
- the allocation of the symbol: register number, displacement in code space, stack displacement, or displacement in heap packet.
- the name of the symbol's associated type in the type map

3.2.3.7.2 The Type Map

The type map provides the map between types and type representations on the target machine. There is one type map file associated with each compilation unit. For each unique type used in a compilation unit, a type descriptor is built which specifies how the type is mapped onto the target machine. The type descriptor is built from default information, if no representation specification for the type was specified; otherwise, built from the representation specification. Each type descriptor is uniquely named (e.g., numbered sequentially starting at 1) so that it may be referenced from the symbol map. An entry for a type consists of:

- the name of the type within the type map
- the node_name of the type's entry in the symbol table
- the type descriptor

3.2.3.8 Updating the Statement Map

The statement map for the compilation unit output by the optimizer is updated when optimizations are performed. The database name of the statement map file to be updated is obtained from the entry for the compilation unit in the library file. The updated statement map is then written to a new file; the original statement map is retained, since it may be needed in other derivations.

3.2.3.9 Outputs

The outputs of the code generator and the default file pathnames are summarized in Table 3-4. Their generation is described in the preceeding sections.

Table 3-4 Outputs of the Code Generator

```
Object File:
     library_file.library_unit{.subunit}.CODE
Symbol map:
     library_file.library_unit{.subunit}.SYMAP
Statement Map:
     library_file.library_unit{.subunit}.SMAP
Type Map:
     library_file.library_unit{.subunit}.TMAP
```

The library file is updated to reflect the new compilation state of the compilation unit. In particular, the database names of the object module and updated statement map file are recorded in the compilation unit's entry.

The pathnames, default or user specified, for the output files in Table 3-4 are also output.

SECTION 4

QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Testing of the Ada compiler shall be in accordance with the schedule, procedures and methods set forth in the following documents:

1. Contractor's Computer Program Development Plan (CPDP)
2. Computer Program test Plan for this CPCI
3. Computer Program Test Procedures for this CPCI.

Testing of the Ada compiler shall be performed at three levels:

1. Computer program component test and evaluation
2. Integration test, involving all components of the CPCI
3. Computer program acceptance testing, involving the APSE

4.1.1 Computer Program Component Test and Evaluation

This level of testing supports development. Each component of the compiler shall be tested as a stand-alone program before integration. This testing shall concentrate on areas where new algorithms have been developed or where there is relatively high risk. Examples of such areas are:

* Expressions
* Resolution of overloading
* Generic expansion
* Separate compilation features

The test bed for unit testing shall be the parts of the compiler that are complete at test time, together with special purpose drivers required for each test.

Test results shall be recorded in informal documentation; formal test reports are not required.

4.1.2 Integration Testing

This level of testing supports integration and prepares for acceptance tests. During this testing, Ada compiler components shall be integrated one at a time and run with previously tested subsets of the complete Ada compiler. Testing at this level shall follow the test plan and procedures for the CPCI. Formal test reports are not required.

4.1.3 Formal Acceptance Testing

This testing assures that the compiler conforms to the language requirements and to requirements in the Type A and B5 specifications. A formal test plan and test procedures shall be generated and used to insure conformance to the requirements. Acceptance tests shall be defined to incrementally test major functional components of the compiler. Acceptance testing shall be witnessed by the Government. Test results shall be documented in accordance with the Computer Program Development Plan and Computer Program Test Plans, and delivered to the Government with final system documentation.

All Ada compilers delivered shall be validated by the Government using the Ada Compiler Validation Facility. Government acceptance of each compiler shall be contingent on the the results of this validation and certification by the Ada Configuration Control Board.

4.2 Test Requirements

Unit testing and integration testing shall be performed using the developed compiler and needed drivers. While testing shall not use formal test plans, testing shall keep the final acceptance tests in mind. Unit tests and integration tests consist of three primary parts: the bootstrap test, execution of the ACVF tests, and special purpose tests designed to exercise compiler features not otherwise adequately covered.

4.2.1 Bootstrap tests

The bootstrap operation consists of compiling the compiler using the compiler itself until the compiler reproduces itself. Three iterations suffice if the compiler has no errors. This is a good check of language features used in the implementation of the compiler, since it tests complex interactions between features that no contrived test could. For example, the bootstrap process tests arithmetic, boolean arithmetic, commonly used control statements, record structures and arrays, since such features are used extensively in compiler construction. The bootstrap test is not adequate, however, for language features rarely used in the implementation of a compiler. Such features would include floating or fixed point arithmetic, tasking, and generics; they require special tests.

4.2.2 Ada Compiler Validation Tests

The DoD Ada Compiler Validation Facility (ACVF) tests are designed to test compiler conformance to the Ada language standard. These tests may also be used for the detection of compiler errors. Each of the ACVF tests is a "pass-fail" trial. The compiler is expected to produce a specific result for each case.

4.2.3 Rehosting tests

Parallel sets of bootstrap, ACVF and special tests shall be run on the IBM 370 and the Interdata 8/32 compilers. Components of the 370 version may be used to simulate or provide drivers for components not yet rehosted on the Interdata 8/32, during unit testing.

4.3 Acceptance Test Requirements

The acceptance tests shall be run according to the contractually developed test plan. The acceptance test shall consist of the three groups of tests described above: the ACVF tests, the compiler bootstrap, and special purpose tests designed to test particular features.

4.3.1 Performance Requirements

The performance of the Ada compiler shall be measured in terms of its use of host system resources and in the efficiency of the software products it generates.

The Government shall specify the machine and operating system configurations for the initial Ada Integrated Environment host systems. Acceptance test plans shall specify compiler performance requirements in terms of processing speed and memory use in these host systems.

The Ada Integrated Environment Statement of Work [RADC80] requires that the delivered compilers shall compile a nontrivial Ada program of at least 50 source statements in at most 256K bytes of memory, at a compilation rate of 1000 statements per minute.

4.4 Independent Validation and Verification

An independent validation and verification (IV&V) contractor, if one participates in the Ada Integrated Environment program, may perform independent testing of the Ada compiler using any of the tests descibed above or additional procedures.

APPENDIX A

THE COMPILER CONTROL LANGUAGE

A.1 Introduction

The compiler control language is the means for specifying the input parameters to the various passes of the Ada optimizing compiler, viz., a library file, the compilation units to be compiled, pragmas to be applied to the compilation and the pathnames of output files. The control language is used to build a command stream for a compilation. A command stream is composed of a specification of a program library, and for each compiler pass, a control sequence for each compilation unit within the program library to be processed by that pass. A compilation unit's control sequence may name files to be used for output generated by the invoked compiler pass and/or may specify language pragmas to be included as an integral part of the compilation unit. Pragmas are applied at the program unit/block level. Therefore, it is possible in a compilation unit's control sequence to associate pragmas with the compilation unit and with a specific program unit/block embedded in the compilation unit.

A command steam specifies the control sequences for one or more compiler passes. However, the compiler passes so specified must be unique.

A.2 Command Language Syntax

The syntax for the compiler control language is specified in Figure A-1.

```
Command_Stream      ::= Library_Designator
                        Compilation_Pass
                        {Compilation_Pass}
Library_Designator  ::= LIBRARY File_Pathname
Compilation_Pass    ::= <<Pass>> BEGIN Control_Sequence
                                       {Control_Sequence} END
Pass                ::= ANALYZER | OPTIMIZER | CODEGEN
Control_Sequence    ::= UNIT Compilation_Unit_Designator
                        {Pragma_Designator}
                        {Output_File_Designator}
                        {Program_Unit_Designator}
Compilation_Unit_Designator
                    ::= File_Pathname
Pragma_Designator   ::= PRAGMA Pragma_Name [(Argument {,Argument})]
Pragma_Name         ::= CONTROLLED | INLINE | INTERFACE |
                        LIST | MEMORY_SIZE | OPTIMIZE | PACK |
                        PRIORITY | STORAGE_UNIT | SUPPRESS |
                        SYSTEM | STATS | OPTLEVEL
Output_File_Designator
                    ::= NAME File_Name_Designator (File_Pathname)
File_Name_Designator
                    ::= SOURCE | SYMTAB | XREF | SSTAT | CSTAT |
                        ENVT | IL | ASMAP | EIL | OIL | OSMAP |
                        CODE | LISTCODE | SYMAP | SMAP | TMAP
Program_Unit_Designator
                    ::= P_UNIT Program_Unit_Name
                        {Pragma_Designator}
Program_Unit_Name   ::= Identifier
File_Pathname       ::= Identifier{.Identifier}
```

Figure A-1 Compiler Command Language Syntax

A.3 Control Language Semantics

A.3.1 The LIBRARY Command

The LIBRARY command designates the pathname of a library file in which the compilation state of the compilation units are maintained. Only one LIBRARY command may appear within a command stream and it must be first.

A.3.2 The UNIT Command

The UNIT command designates a specific compilation unit to be compiled. Further commands in the command stream pertain to the designated compilation unit until another UNIT command is encountered. The parameter to this command is the pathname of a source file when the analyzer pass is being invoked, and the pathname of the appropriate IL file when the expander, optimizer, or code generator pass is invoked.

A.3.3 The PRAGMA Command

The PRAGMA command permits language pragmas to be designated from outside the Ada program source text. This capability serves two purposes: 1) pragmas may supplement those already in the source (for example, the OPTIMIZE pragma may be introduced); and 2) pragmas which already exist in the source text may be overridden (for example, the source listing may be turned on/off). It is as if the pragma had occurred in the source text in the position as defined for the pragma. The one exception is LIST which is positioned before the named unit. The acceptable pragmas which may appear are the predefined language pragmas, except INCLUDE, (cf. [DoD80B], Appendix B) and any implementation defined pragmas.

A.3.4 The NAME Command

The Name command designates the pathnames of output files that may be generated by the compiler pass when processing a compilation unit. Output files designated by the NAME command must be relevant to the compiler pass; otherwise they are ignored. When a file is not specified in a NAME command, a default name is used (cf. Section 3.1.1.4, Section 3.2.2 Section 3.2.3). The output files which may be designated are summarized in Table A-1.

Table A-1 Output File Designators

File_Name_Designator	Output File	Compiler Pass
DIANA	Generated IL	analyzer
ASMAP	Statement Map	analyzer
OIL	Optimized IL	exp/opt
OSMAP	Updated Statement Map	exp/opt
CODE	Object Module	code generator
SYMAP	Symbol Map	code generator
SMAP	Updated Statement Map	code generator
TMAP	Type Map	code generator

A.3.5 The P_UNIT Command

The P_UNIT command designates individual program units/blocks within the compilation unit to which following PRAGMA commands are to be applied. Further commands in the command stream pertain to the designated program unit/block until another P_UNIT or UNIT is encountered.

A.4 Command Stream Examples

The examples below are for the case where each compiler pass has its own control file and is invoked disjointly. The block of control sequences can be combined into one command stream with only one LIBRARY command.

Example A-1 illustrates a typical command stream provided to the Ada compiler when the analyzer pass is activated (indentation is for readability purposes). In this example, the compilation unit will be a source file since this is an analyzer command stream. The source file, 'Core.Source' is a compilation unit within the 'Alpha.Beta.Project' program library. Specified pragmas will be applied, both to the compilation unit as a whole and to named program units within the command sequence. Space optimization is specified for the entire compilation unit. The output file for the generated IL will be 'Core.Diana'.

```
LIBRARY Alpha.Beta.Project
<<ANALYZER>> BEGIN
   UNIT Core.Source
      PRAGMA List (On)
      PRAGMA Optimize (Space)
      NAME DIANA (Core.Diana)
   P_UNIT GETNAME
      PRAGMA List (Off)
   P_UNIT STORECORE
      PRAGMA Inline
   END
```

Example A-1 Compiler Command Stream for the Analyzer

Example A-2 illustrates a typical command sequence to control the operation of the expander/optimizer pass of the compiler. The compilation unit designator is the pathname of the DIANA file generated by the analyzer (Example A-1), viz., 'Core.Diana'. The level of machine independent optimization is specified to be 2. The expanded/optimized DIANA will be output to the named file, 'Core.Optil'.

```
LIBRARY Alpha.Beta.Project
<<OPTIMIZER>> BEGIN
    UNIT Core.Diana
       PRAGMA Optlevel (2)
       NAME OIL (Core.Optil)
    END
```

Example A-2 Compiler Command Stream for the Expander/Optimizer

Example A-3 illustrates a typical command sequence to control the operation of the code generator pass of the compiler. The compilation unit designator is the pathname of the optimized IL generated by the expander/optimizer (Example A-2), viz., 'Core.Optil'. The generated code will be output to the named file 'Core.Objcode'. The command sequence also specifies the pathnames of files generated by the code generator, viz., a symbol map, 'Core.Symap', type map, 'Core.Typemap', an updated statement map, and a 'Core.Stmtmap' for use by the source level debugger.

```
LIBRARY Alpha.Beta.Project
<<CODEGEN>> BEGIN
    UNIT Core.Optil
       NAME Code (Core.Objcode)
       NAME Symap (Core.Symap)
       NAME Smap (Core.Stmtmap)
       NAME Tmap (Core.Typemap)
    END
```

Example A-3 Compiler Command Stream for the Code Generator

APPENDIX B

A RECURSIVE DESCENT PARSER FOR ADA

B.1 Construction of the Recursive Descent Parser

The grammar was run through a syntax analyzer to determine whether or not the grammar is LL(1). If the grammar was LL(1), it would have been possible to construct a completely factored parser which would execute as efficiently as a table driven parser. Since that grammar was not LL(1), special techniques must be used in some cases, especially where the grammar is ambiguous.

One possible solution is to "look ahead" one or two symbols. This usually helps only in simple cases; however, when it can be used it is probably the simplest method. More difficult cases are handled by writing composite routines which essentially recognize more than one alternative simultaneously by remembering what was scanned. Finally, for the cases where there is ambiguity, a composite routine will make use of semantic information to r⁻:olve the ambiguity. More detailed examples of these techniques follow.

A side benefit of the syntax analyzer is the determination of selection sets for each production. The LL(1) criteria is basically that all of the selection sets for a given production be distinct. Therefore, examination of the syntax analyzer output will permit systematic construction of the parser. The analyzer output also indicates where clashes exist when the LL(1) condition is not satisfied to help in determining how to handle problems when they arise.

The output of the grammar analyzer is shown in Appendix C. This listing consists of several parts. The first part is simply the listing of the grammar for Ada. It was run on an IBM 370, therefore some character translations were necessary. The listing of the grammar is followed by a listing of all of the terminal symbols of the grammar followed by all of the nonterminal symbols of the grammar. Since the grammar analyzer actually handles standard BNF, productions starting with "X" followed by two lower case letters are generated for all productions enclosed in square or curley brackets. These listings are followed by a listing of the productions of the grammar with the select set for each production and an indication of whether or not that particular production satisfies the LL(1) criteria.

B.1.1 The Basic Approach

The basic approach used is to represent the Ada syntax as syntax diagrams constructed from the modified grammar, and then implement the syntax diagrams as recursive subroutines. In a simplistic implementation, there would be a syntax diagram for each non-terminal of the grammar. In reality, this probably is not the case since this would result in modules which were much too small. Simple productions shall be subsumed into higher level productions, i.e., reduced syntax diagrams shall be obtained by suitable substitution of non-terminals in diagrams.

For example, the production for 'alignment_clause' consists of two terminals, AT and MOD, followed by a 'simple_expression'. 'Alignment_clause' is only used in one production. Therefore, rather than have a separate routine, the 'alignment_clause'; production shall simply be recognized in the production for 'record_type_representation'. This is purely for efficiency and does not affect the concept of a routine for every non-terminal symbol. Since this part of the design is essentially well understood, the design will not be carried to the level of detail of each routine. Examples will be given for a typical routine, and special attention paid to those areas where there are problems with the grammar.

In addition to syntactic analysis, the recursive routines shall also perform semantic analysis and actually produce the IL, i.e., the AST. For example, the syntax for 'if_statement' is:

```
if_statement ::=
    IF condition THEN sequence_of_statements
    {ELSIF condition THEN sequence_of_statements}
    [ELSE sequence_of_statements]
    END IF;
```

The corresponding syntax diagram is:

```
--->IF-->condition----->THEN--->sequence_of_statements-------+
                                                             |
  +----------------------------------------------------------+
  |  +------------------------------------------------+
  |  |                                                |
  |  |                                                V
  +----->ELSIF--->condition-->sequence_of_statements------+
     A                                                    |
     |                                                    |
     +----------------------------------------------------+
                                                          |
 +--------------------------------------------------------+
 |  +----------------------------------------------+
 |  |                                              |
 |  |                                              V
 +--+->ELSE--->condition-->sequence_of_statements----->END--+
                                                            |
                                             <---;<---IF<---+
```

The syntax routine developed from this syntax diagram would perform the following actions:

1. The terminal symbol IF has already been recognized before entry to this routine by the calling routine. An IL node to represent the 'if_statement' is created, and attrib ·:s filled in later.

2. Call the routine for condition, if successful it returns a pointer to the IL node which represents the root of the AST for that condition. Fill in the corresponding field in the 'if_statement' IL node. If unsuccessful, call the routine SKIP which skips to a symbol in the error set for this routine, and output an appropriate error message.

3. Recognize a 'sequence_of_statements'. (This is terminated by ELSE, ELSIF or END). Fill in the field corresponding to the 'sequence_of_statements' in the 'if_statement' IL node created earlier to reference the AST for the 'sequence_of_statements'.

4. Recognize the terminal symbol THEN. If not present output an error message and SKIP.

5. Recognize ELSIF. If not present go to step 6. Recognize condition as in step 2. Recognize a 'sequence_of_statements'. Construct an IL node to represent the elsif part and fill in the field of either the previous if or elseif node. Note this is dependent to some extent upon the IL chosen. Go to step 4.

6. Recognize ELSE. If present recognize a 'sequence_of_statements' and link it in as the a field of the preceeding node.

7. Recognize END, followed by IF. IF not present, SKIP and output an appropriate error message. The tree represented by the IL node for the 'if_statement' is returned as a reference either through a global variable, a parameter, or possibly as a function result. In any case, this is an implementation detail.

This pseudocode description is long primarily because it is trying to be descriptive. The actual code will probably be comparable in size, and in some ways, much clearer. Many of the steps which are done repetitively and in a large number of places will be made into single procedures.

The 'if_statement' routine is typical of the implementation of the syntactic routines for most of the Ada grammar. Implementation of routines from syntax diagrams for other language constructs proceeds in a similar straigt-forward process. Typical areas where problems occur are discussed below. Problem areas are those where the grammar is not LL(1). Note, if the grammar were LL(1), the entire implementation would be as simple as the example described above. Areas where the grammar diverges from LL(1) are easily fixed by simple rewriting of the grammar; in others, where the grammar is actually ambiguous more obscure techniques are required.

B.1.2 Left Factoring

One simple technique which can be used both for efficiency and to make the productions LL(1) is left factoring. For example, the production for 'renaming_declaration' is as follows:

```
renaming_declaration ::=
        identifier : type_mark RENAMES name;
      | identifier : EXCEPTION RENAMES name;
      | PACKAGE identifier RENAMES name;
      | TASK identifier RENAMES name;
```

This particular production is not LL(1) because identifier begins each of the first two alternatives. In this case, there is not a real problem, because the following productions are clearly equivalent:

```
renaming_declaration ::=
        identifier : new_production RENAMES name
      | PACKAGE identifier RENAMES name;
      | TASK identifier RENAMES name;
new_production ::= type_mark | EXCEPTION
```

Since EXCEPTION is a terminal symbol and is distinct from the set of terminal symbols which can begin 'type_mark', this new production satisfies the LL(1) condition. A syntax diagram of the procedure to implement this production might be as follows:

```
renaming_declaration

                +-->type_mark---+
                |               |
                |               V
--->identifier----->EXCEPTION------->RENAMES---->name---->;-->
 |                              A
 |                              |
 +-->PACKAGE---->identifier-----+
 |                              |
 +-->TASK------->identifier-----+
```

Note, in this diagram the RENAMES, 'name', and ";" are factored. This is more efficient to code because there is less repeated code, but the syntax routine must remember whether the result is a type, an exception, a package or a task. This presents no problem. The result is either a field of the IL node created early in the routine or can be remembered in a variable.

B.1.3 Composition

Now consider an example where left factoring does not work as well, namely the productions:

```
allocator ::=  NEW type_mark [(expression)]
             | NEW type_mark aggregate
             | NEW type_mark discriminate_constraint
             | NEW type_mark index_constraint
```

The productions for allocator are not LL(1) since they all start with "NEW". However, by simply introducing a new production and left factoring out the common part, this problem goes away.

```
allocator ::= NEW type_mark new_prod
new_prod  ::= aggregate | discriminant_constraint
            | index_constraint | [(expression)]
```

While this solves the problem at this level, new problems occur because the select sets for the alternatives of 'new_prod' are not unique and there is nothing that can be simply factored out. In this case, a composite routine shall be written which simultaneously scans for one of the allowable four productions. This production may have the capability to lookahead some number of symbols to differentiate between the possible alternatives before deciding whihc routine to call. The productions of interest are:

```
aggregate ::= (component_association {, component_association})
component_association ::= [choice {"|" choice} =>] expression
choice ::= simple_expression | discrete_range  | OTHERS
discrete_range ::= type_mark [range_constraint] | range
range ::= simple_expression .. simple_expression
discriminant_constraint ::=
   (discriminant_specification {, discriminant_specification})
discriminant_specification ::=  [name {"|" name} =>] expression
index_constraint ::= (discrete_range {, discrete_range})
type_mark ::= name
```

In this case, the problem is more severe since this actually represents an ambiguous grammar. Note, for some input, 'aggregate' reduces to 'expression' which reduces to "(expression)". Also 'discriminant_constraint' reduces to "(expression)". Therefore, for the input string: "(id)" syntactically there are at least three different parse trees:

```
              new_prod                              new_prod
                  |                                     |
        discriminant_constraint                     aggregate
                  |                                     |
   +------------------------------+       +-----------------------+
   |              |               |       |           |           |
   (  discriminant_specification )        ( component_association )
                  |                                   |
             expression                           expression

                              new_prod
                                  |
                          +--------------+
                          |      |       |
                          (  expression  )
```

To resolve this ambiguity, a composite routine shall be written to take advantage of available semantic information. In this case, the differentiating semantics is the previously recognized type, and the semantic rules concerning the make up of each of these components. For example, semantically, aggregates with only one element must be given in the named notation. There are similar rules about discriminants. Further, trouble is encountered when expression is considered and anticipating that trouble, 'expression' will eventually be constrained to return the type associated with the expression. Using this semantic information, a composite routine can be written to handle this production.

B.1.4 Combination of Non-terminal Productions

Another problem occurs with the production 'array_type_definition'. Relevant productions are:

```
array_type_definition ::=
      ARRAY (index {, index}) OF subtype_indication
    | ARRAY index_constraint OF subtype_indication
index_constraint ::=  (discrete_range {, discrete_range})
index ::= type_mark RANGE <>
discrete_range ::= type_mark [range_constraint] | range
range ::= simple_expression .. simple_expression
range_constraint ::= RANGE range
```

ARRAY can be left factored, however, there are still problems since 'index_constraint' can begin with "(". Bringing the definition of 'index_constraint' up a level results in a rewritten grammar:

```
array_type_definition ::=
     ARRAY (newprod) OF subtype_indication
newprod ::= newprodl {, newprodl}
newprodl::= index | discrete_range
```

A simple routine shall be written which handles this. Application of the semantic rule that 'index' and 'discrete_range' cannot be mixed in the list results in the same language being parsed. Note, the routine for 'newprodl' must scan to the box to resolve which alternative is to be recognized.

B.1.5 Incorporating Semantics

Another technique to eliminate problems with the grammar is to modify the grammar and use a semantic rule which cannot be expressed in BNF. For example, the production for 'expression' is:

```
expression ::=  relation {AND relation}
              | relation {OR  relation}
              | relation {XOR relation}
              | relation {AND THEN relation}
              | relation {OR ELSE  relation}
```

This creates problems because each alternative begins with relation. This production requires parenthesis if the relational operators are mixed. The following productions are equivalent if combined with a semantic rule that all non-parenthesized occurrences of 'bool_op' must be the same literal string.

```
expression ::= relation {bool_op relation}
   bool_op ::= AND | OR | XOR | AND THEN | OR ELSE
```

There is still a minor problem in differentiating between AND and AND THEN, and between OR and OR ELSE. This can be resolved simply by looking ahead a single symbol.

B.1.6 Elimination of Left Recursion

There are also multiple problems with the productions for 'name'. The relevant productions are:

```
name ::= identifier | indexed_component | slice
      | selected_component | attribute | function_call
      | operator_symbol
indexed_component ::= name (expression {, expression})
slice ::= name (discrete_range)
selected_component ::= name.identifier | name.ALL
                     | name.operator_symbol
attribute ::= name'identifier
function_call ::= name actual_parameter_part | name( )
discrete_range ::= type_mark [range_constraint]
                 | simple_expression .. simple_expression
```

In addition to not being LL(1), 'name' is also indirectly left recursive. For example, 'indexed_component', 'selected_component', 'attribute' and 'function_call' all begin with 'name'. Recursive descent parsers do not tolerate left recursion. There are algorithms for eliminating left recursion that could be applied; they eliminate left recursion by changing to iteration or right recursion. To convert to iteration, the rule is:

```
Transform a left recursive production of the form,
 "a::=a x|b" to an iteration of the form "a::=b { x}"
```

Application of this rule to 'name' after applying left factoring gives:

```
name ::= name2 {name1}
name1 ::= (expression {, expression}) | (discrete_range)
        | .name3
        | 'identifier | actual_parameter_part | ( )
name2 ::= identifier | operator_symbol
name3 ::= identifier | ALL | operator_symbol
```

A composite routine shall be written to process these productions. Semantic information shall be used to resolve similarities between 'expression', "discrete_range' and 'actual_parameter_part'.

APPENDIX C

A PRACTICAL GRAMMAR FOR ADA

C.1 Ada Concrete Syntax

```
sequence_of_statements ::= statement { statement }

statement ::=
   { label } simple_statement | { label } compound_statement

simple_statement ::= null_statement
   | assignment_statement | exit_statement
   | return_statement     | goto_statement
   | procedure_call       | entry_call
   | delay_statement      | abort_statement
   | raise_statement      | code_statement

compound_statement ::=
     if_statement         | case_statement
   | [ identifier ":" ] compound_statement1
   | accept_statement     | select_statement

compound_statement1 ::= loop_statement | block

label ::= "<<" identifier ">>"

null_statement ::= NULL ;

assignment_statement ::= name ":=" expression ;

if_statement ::=
   IF condition THEN sequence_of_statements
  { ELSIF condition THEN sequence_of_statements }
  [ ELSE sequence_of_statements ]
   END IF ;

condition ::= expression

case_statement ::=
   CASE expression IS
     { WHEN choice { "|" choice } => sequence_of_statements }

loop_statement ::=  [ iteration_clause ] basic_loop [ identifier ] ;

basic_loop ::=  LOOP sequence_of_statements END LOOP
```

```
iteration_clause ::=
   FOR loop_parameter IN [ REVERSE ] discrete_range | WHILE condition

loop_parameter ::= identifier

block ::=
   [ DECLARE
       declarative_part ]
    BEGIN
       sequence_of_statements
   [ EXCEPTION { exception_handler } ]
    END [ identifier ] ;

exit_statement ::=
   EXIT [ name ] [ WHEN condition ] ;

return_statement ::= RETURN [ expression ] ;

goto_statement ::= GOTO name ;

choice ::= simple_expression | discrete_range | OTHERS

declarative_part ::=  { declarative_item }
   { representation_specification } { program_component }

declarative_item ::= declaration | use_clause

accept_statement ::=
   ACCEPT name [ formal_part ] [ DO
     sequence_of_statements
   END [ identifier ] ] ;

delay_statement ::= DELAY simple_expression ;

select_statement ::= selective_wait
   | conditional_entry_call | timed_entry_call

selective_wait ::=
     SELECT
       [ WHEN condition => ]
          select_alternative
   { OR [ WHEN condition => ]
          select_alternative }
   [ ELSE
     sequence_of_statements ]
   END SELECT ;

select_alternative ::=
     accept_statement [ sequence_of_statements ]
   | delay_statement  [ sequence_of_statements ]
   | TERMINATE ;
```

```
conditional_entry_call ::=
   SELECT
     entry_call [ sequence_of_statements ]
   ELSE
     sequence_of_statements
   END SELECT ;

timed_entry_call ::=
   SELECT
     entry_call [ sequence_of_statements ]
   OR
     delay_statement [ sequence_of_statements ]
   END SELECT ;

abort_statement ::= ABORT name { , name } ;

raise_statement ::= RAISE [ name ] ;

range_constraint ::= RANGE range

range ::= simple_expression .. simple_expression

discrete_range ::= type_mark [ range_constraint ] | range

use_clause ::= USE name { , name } ;

actual_parameter_part ::=
   ( parameter_association { , parameter_association } )

parameter_association ::=
   [ formal_parameter => ] actual_parameter

formal_parameter ::= identifier

actual_parameter ::= expression

procedure_call ::=
   name [ actual_parameter_part ] ;

exception_handler ::=
   WHEN exception_choice { "|" exception_choice } =>
     sequence_of_statements

exception_choice ::= name | OTHERS

entry_call ::= name [ ( actual_parameter_part ) ] ;

code_statement ::= qualified_expression ;

identifier ::= letter { [ underscore ] letter_or_digit }

letter_or_digit ::= letter | digit
```

```
letter ::= upper_case_letter | lower_case_letter

numeric_literal ::= decimal_number | based_number

decimal_number ::= integer [ . integer ] [ exponent ]

integer ::= digit { [ underscore ] digit }

exponent ::= E exponent1

exponent1 ::=  [ + ] integer |  - integer

based_number ::= base # based_integer [ . based_integer ] # [ exponent ]

base ::= integer

based_integer ::= extended_digit { [ underscore ] extended_digit }

extended_digit ::= digit | letter

character_string ::= "{ character }"

function_call ::=
  name actual_parameter_part | name ( )

name ::= identifier
   | indexed_component | slice
   | selected_component | attribute
   | function_call | operator_symbol

indexed_component ::= name ( expression { , expression } )

slice ::= name ( discrete_range )

selected_component ::=  name .  selected_compl

selected_compl ::= identifier | ALL | operator_symbol

attribute ::= name "'" identifier

literal ::=
   numeric_literal | enumeration_literal | character_string |NULL

aggregate ::=
   ( component_association { , component_association } )

component_association ::=
   [ choice { "|" choice } => ] expression

expression ::=
     relation { AND  relation }
   | relation { OR relation }
   | relation { XOR relation }
```

```
    | relation {  AND THEN relation }
    | relation { OR ELSE relation }

  relation ::=
     simple_expression [ relational_operator simple_expression ]
      | simple_expression [ NOT ] IN range
      | simple expression [ NOT ] IN subtype_indication

   simple_expression ::= [ unary_operator ] term { adding_operator term }

   term ::= factor { multiplying_operator factor }

   factor ::= primary [ ** primary ]

  primary ::=
     literal | aggregate | name | allocator | function_call
    | type_conversion | qualified_expression | ( expression )

  logical_operator          ::= AND | OR | XOR

  relational_operator       ::= =   | /= | "<"  | "<="  | ">"  | ">="

  adding_operator           ::= +   | -  | &

  unary operator            ::= +   | -  | NOT

  multiplying_operator      ::= *   | /  | MOD | REM

  exponentiating_operator   ::= **

  type_conversion ::= type_mark ( expression )

  qualified_expression ::= typemark qualified_exl

  qualified_exl ::= ( expression ) | "'" aggregate

  allocator ::= NEW type_mark allocatorl

  allocatorl ::= [ ( expression ) ] | aggregate | discriminant_constraint
               | index_constraint

  declaration ::= declarationl | type_declaration
      | subtype_declaration | subprogram_declaration | package_declaration
      | task declaration

  declarationl ::= identifier_list ":" declaration2

  declaration2 ::= object_or_number_declaration
                 | EXCEPTION ; | renaming_declaration

  object_or_number_declaration ::=
       [ CONSTANT ] subtype_indication       [ ":=" expression ] ;
    |  [ CONSTANT ] array_type_definition    [ ":=" expression ] ;
   |    CONSTANT ":=" expression ;

 identifier_list ::= identifier { , identifier }

 type_declaration ::= TYPE identifier [ discriminant_part ]
         IS type_definition ;      | incomplete_type_declaration

 type_definition ::= enumeration_type_definition | integer_type_definition
  | real_type_definition | array_type_definition | record_type_definition
  | access_type_definition | derived_type_definition | private_type_definition

 subtype_declaration ::= SUBTYPE identifier IS subtype_indication

 subtype_indication ::= type_mark [ constraint ]

 type_mark ::= name

 constraint ::= range_constraint | accuracy_constraint | index_constraint |
        discriminant_constraint

 derived_type_definition ::=  NEW subtype_indication

 enumeration_type_definition ::=
  ( enumeration_literal { , enumeration_literal } )

 enumeration_literal ::= identifier | character_literal

 integer_type_definition ::= range_constraint

 real_type_definition ::= accuracy_constraint

 accuracy_constraint ::= floating_point_constraint | fixed_point_constraint

 floating_point_constraint ::=
           DIGITS simple_expression [ range_constraint ]

 fixed_point_constraint ::=
       DELTA simple_expression [ range_constraint ]

 array_type_definition ::=
   ARRAY ( index { , index } ) OF subtype_indication
   | ARRAY index_constraint  OF subtype_indication
   ARRAY ( index1 { , index1 } ) OF subtype_indication

 index1 ::= type_mark RANGE "<>"
         | type_mark [ range_constraint ]  | range

 index_constraint ::= ( discrete_range { , discrete_range } )

 record_type_definition ::= RECORD component_list END RECORD

 component_list ::= { component_declaration } [ variant_part ] | NULL ;
```

```
component_declaration ::=
    identifier_list ":" subtype_indication  [ ":=" expression ] ;
  | identifier_list ":" array_type_definition [ ":=" expression ] ;

discriminant_part ::=
    ( discriminant_declaration { ; discriminant_declaration } )

discriminant_declaration ::=
   identifier_list ":" subtype_indication [ ":=" expression ]

discriminant_constraint ::=
   ( discriminant_specification { , discriminant_specification } )

discriminant_specification ::= [ name { "|" name } => ] expression

variant_part ::=
  CASE name IS
    { WHEN choice { "|" choice } => component_list }  END CASE ;

access_type_definition ::= ACCESS subtype_indication

incomplete_type_declaration ::= TYPE identifier [ discriminant_part ] ;

pragma ::=  PRAGMA  identifier [ ( argument { , argument } ) ] ;

argument ::= [ identifier => ] argument1

argument1 ::= name | expression

program_component ::= body
    | package_declaration | task_declaration | body_stub

body ::= subprogram_body | package_body | task_body

subprogram_declaration ::= subprogram_specification ;
   | generic_subprogram_declaration
   | generic_subprogram_instantiation

subprogram_specification ::=
     PROCEDURE identifier [ formal_part ]
   | FUNCTION designator  [ formal_part ] RETURN subtype_indication

designator ::= identifier | operator_symbol

operator symbol ::= character_string

formal_part ::=
   ( parameter_declaration { ; parameter_declaration } )

parameter_declaration ::=
   identifier_list ":" mode subtype_indication [ ":=" expression ]

mode ::= [ IN ] | OUT | IN OUT
```

```
subprogram_body ::=
   subprogram_specification IS
     declarative_part
   BEGIN
     sequence_of_statements
   [ EXCEPTION
     {  exception_handler } ]
   END [ designator ] ;

package_declaration ::= generic_package_declaration
               | package_decl_header package_decl

package_decl_header ::= PACKAGE identifier IS

package_decl ::= package_specification | generic_package_instantiation

package_specification ::=
     { declarative_item }
   [ PRIVATE  { declarative_item }
     { representation_specification } ]
   END [ identifier ]

package_body ::=
   PACKAGE BODY identifier IS
     declarative_part
   [ BEGIN
     sequence_of_statements
   [ EXCEPTION
     { exception_handler } ] ]
   END [ identifier ] ;

private_type_definition ::= [ LIMITED ] PRIVATE

renaming_declaration ::=
    renaming_declaration1 RENAMES name ;
  | PACKAGE identifier RENAMES name ;
  | TASK    identifier RENAMES name ;
  | subprogram_specification RENAMES name ;

renaming_declaration1 ::= type_mark | EXCEPTION

task_declaration ::= task_specification

task_specification ::=
   TASK [ TYPE ] identifier [ IS
     { entry_declaration }
     { representation_specification }
   END [ identifier ] ] ;

task_body ::=
   TASK BODY identifier IS
     [ declarative_part ]
     BEGIN
       sequence_of_statements
     [ EXCEPTION
       { exception_handler } ]
     END [ identifier ] ;

  entry declaration ::=
     ENTRY identifier [ ( discrete_range ) ] [ formal_part ] ;

  compilation ::= { compilation_unit }

  compilation_unit ::= context_specification compilation_unitl

  compilation_unitl ::=      subprogram_declaration
     |  subprogram_body | package_declaration | package_body | subunit

  context specification ::= { with_clause [ use_clause ] }

  with clause ::= WITH name { , name } ;

  subunit ::=
    SEPARATE ( name ) subunit_body

  subunit body ::=
     subprogram_body | package_body | task_body

  body stub ::=
       subprogram_specification IS SEPARATE ;
     | PACKAGE BODY identifier  IS SEPARATE ;
     | TASK BODY identifier     IS SEPARATE ;

  generic_subprogram_declaration ::=
        generic_part  subprogram_specification ;

  generic_package_declaration ::=
      generic_part package_specification ;

  generic_part ::= GENERIC { generic_formal_parameter }

  generic formal_parameter ::=
       parameter_declaration ;
     | TYPE identifier [ discriminant_part ] IS generic_type_definition ;
     | WITH subprogram_specification [ IS name ] ;
     | WITH subprogram_specification IS "<>" ;

  generic type_definition ::=
       ( "<>" ) | RANGE "<>" | DELTA "<>" | DIGITS "<>"
       | array_type_definition | access_type_definition
       | private_type_definition

  generic subprogram_instantiation ::=
       PROCEDURE identifier IS generic_instantiation ;
     | FUNCTION designator  IS generic_instantiation ;
```

```
generic_package_instantiation ::=
   NEW name [ ( generic_association { , generic_association } ) ]

generic_association ::=
   [ formal_parameter => ] generic_actual_parameter

generic_actual_parameter ::=
   expression | name | subtype_indication

representation_specification ::=
     length_specification       | enumeration_type_representation
   | record_type_representation | address_specification

length_specification ::= FOR attribute USE expression ;

enumeration_type_representation ::= FOR name USE aggregate ;

record_type_representation ::=
   FOR name USE
     RECORD [ alignment_clause ; ]
       { name location ; }
     END RECORD ;

location ::= AT simple_expression RANGE range

alignment_clause ::= AT MOD simple_expression

address_specification ::= FOR name USE AT simple_expression ;
```

C.2 Productions

```
abort_statement   ::=
     ABORT name Xbh ;
                    ==>ABORT

accept_statement   ::=
     ACCEPT name Xao Xap ;
                    ==>ACCEPT

access_type_definition   ::=
     ACCESS subtype_indication
                    ==>ACCESS

accuracy_constraint   ::=
     floating_point_constraint  |
                    ==>DIGITS
     fixed_point_constraint
                    ==>DELTA
```

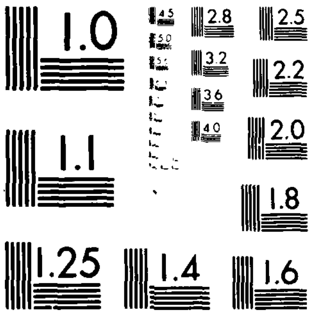
```
actual parameter   ::=
     expression
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

actual parameter_part   ::=
     ( parameter_association Xb  )
                    ==>(

adding operator   ::=
     +  |
                    ==>+
     -  |
                    ==>-
     &
                    ==>&

address_specification   ::=
     FOR name USE AT simple_expression ;
                    ==>FOR

aggregate   ::=
     ( component_association Xcc )
                    ==>(

alignment clause   ::=
     AT MOD simple_expression
                    ==>AT

allocator   ::=
     NEW type_mark allocator1
                    ==>NEW

allocator1   ::=
     Xck  |
                    ==>.. ( + & * ** ) ; - / /= , = => "<="
                    ==>"<" "|" ">=" ">" ":=" lower_case_letter
                    ==>upper_case_letter AND ARRAY BEGIN END
                    ==>FOR FUNCTION GENERIC IN IS LOOF MOD NOT
                    ==>OR PACKAGE PRIVATE PROCEDURE RANGE REM
                    ==>RENAMES SUBTYPE TASK THEN TYPE USE XOR
                    ==>
     aggregate  |
     ***NOT LL(1)***
                    ==>(
     discriminant_constraint  |
     ***NOT LL(1)***
                    ==>(
     index constraint
     ***NOT LL(1)***
                    ==>(
argument   ::=
     Xdn argument1
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

argument1   ::=
     name  |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     expression
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

array_type_definition   ::=
     ARRAY ( index Xdc ) OF subtype_indication  |
                    ==>ARRAY
     ARRAY index_constraint OF subtype_indication ARRAY ( index1 Xdd ) OF
          subtype_indication
     ***NOT LL(1)***
                    ==>ARRAY

assignment_statement   ::=
     name ":=" expression ;
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

attribute   ::=
     name "'" identifier
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

base   ::=
     integer
                    ==>digit

based_integer   ::=
     extended_digit Xb
                    ==>digit lower_case_letter upper_case_letter
                    ==>

based_number   ::=
     base # based_integer Xbq # Xbr
                    ==>digit

basic_loop   ::=
     LOOP sequence_of_statements END LOOP
                    ==>LOOP

block   ::=
     Xa  BEGIN sequence_of_statements Xa  END Xa  ;
                    ==>BEGIN DECLARE

body   ::=
     suhprogram_body  |
                    ==>FUNCTION PROCEDURE
     package body  |
                    ==>PACKAGE
     task_body
                    ==>TASK

body_stub   ::=
     subprogram_specification IS SEPARATE ;  |
                    ==>FUNCTION PROCEDURE
     PACKAGE BODY identifier IS SEPARATE ;  |
                    ==>PACKAGE
     TASK BODY identifier IS SEPARATE ;
                    ==>TASK

case statement   ::=
     CASE expression IS Xag
                    ==>CASE

character_string   ::=
     "{ character }"
                    ==>"{ character }"

choice   ::=
     simple_expression  |
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     discrete_range  |
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     OTHERS
                    ==>OTHERS

code_statement   ::=
     qualified_expression ;
                    ==>typemark

compilation   ::=
     Xer
                    ==>FUNCTION GENERIC PACKAGE PROCEDURE SEPARATE
                    ==>WITH

compilation_unit   ::=
     context_specification compilation_unitl
                    ==>FUNCTION GENERIC PACKAGE PROCEDURE SEPARATE
                    ==>WITH
```

```
compilation_unitl   ::=
     subprogram_declaration  |
                    ==>FUNCTION GENERIC PROCEDURE
     subprogram_body  |
     ***NOT LL(1)***
                    ==>FUNCTION PROCEDURE
     package_declaration  |
     ***NOT LL(1)***
                    ==>GENERIC PACKAGE
     package_body  |
     ***NOT LL(1)***
                    ==>PACKAGE
     subunit
                    ==>SEPARATE

component_association   ::=
     Xcd expression
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL OTHERS

component_declaration   ::=
     identifier_list ":" subtype_indication Xdi ;  |
                    ==>lower_case_letter upper_case_letter
     identifier_list ":" array_type_definition Xd  ;
     ***NOT LL(1)***
                    ==>lower_case_letter upper_case_letter

component_list   ::=
     Xdg Xdh  |
                    ==>lower_case_letter upper_case_letter CASE
                    ==>END WHEN
     NULL ;
                    ==>NULL

compound_statement   ::=
     if_statement  |
                    ==>IF
     case_statement  |
                    ==>CASE
     Xad compound_statementl  |
                    ==>lower_case_letter upper_case_letter BEGIN
                    ==>DECLARE FOR LOOP WHILE
     accept_statement  |
                    ==>ACCEPT
     select_statement
                    ==>SELECT

compound_statementl   ::=
     loop_statement  |
                    ==>FOR LOOP WHILE
     block
```

```
                    ==>BEGIN DECLARE

condition   ::=
      expression
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

conditional_entry_call   ::=
      SELECT entry_call Xbe ELSE sequence_of_statements END SELECT ;
                    ==>SELECT

constraint   ::=
      range_constraint  |
                    ==>RANGE
      accuracy_constraint  |
                    ==>DELTA DIGITS
      index_constraint  |
                    ==>(
      discriminant_constraint
      ***NOT LL(1)***
                    ==>(

context_specification   ::=
      Xe
                    ==>FUNCTION GENERIC PACKAGE PROCEDURE SEPARATE
                    ==>WITH

decimal_number   ::=
      integer Xbl Xbm
                    ==>digit

declaration   ::=
      declaration1  |
                    ==>lower_case_letter upper_case_letter
      type_declaration  |
                    ==>TYPE
      subtype_declaration  |
                    ==>SUBTYPE
      subprogram_declaration  |
                    ==>FUNCTION GENERIC PROCEDURE
      package_declaration  |
      ***NOT LL(1)***
                    ==>GENERIC PACKAGE
      task_declaration
                    ==>TASK

declaration1   ::=
      identifier_list ":" declaration2
                    ==>lower_case_letter upper_case_letter

declaration2   ::=
      object_or_number_declaration  |
```

AD-A109 978 TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP F/G 9/2
ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPE--ETC(U)
DEC 81 F30602-80-C-0293
UNCLASSIFIED RADC-TR-81-360-VOL-3 NL

2 OF 3
AD
A109978


1.0
4.5
5.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

```
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>ARRAY CONSTANT
     EXCEPTION ;  |
                    ==>EXCEPTION
     renaming_declaration
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>EXCEPTION FUNCTION PACKAGE PROCEDURE
                    ==>TASK

declarative_item   ::=
     declaration  |
                    ==>lower_case_letter upper_case_letter FUNCTION
                    ==>GENERIC PACKAGE PROCEDURE SUBTYPE TASK
                    ==>TYPE
     use_clause
                    ==>USE

declarative_part   ::=
     Xal Xam Xan
                    ==>lower_case_letter upper_case_letter BEGIN
                    ==>END FOR FUNCTION GENERIC PACKAGE PROCEDURE
                    ==>SUBTYPE TASK TYPE USE

delay_statement   ::=
     DELAY simple_expression ;
                    ==>DELAY

derived_type_definition   ::=
     NEW subtype_indication
                    ==>NEW

designator   ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     operator_symbol
                    ==>"{ character }"

discrete_range   ::=
     type_mark Xb   |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     range
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

discriminant_constraint   ::=
     ( discriminant_specification Xd  )
                    ==>(

discriminant_declaration   ::=
     identifier_list ":" subtype_indication Xd
                    ==>lower_case_letter upper_case_letter

discriminant_part   ::=
     ( discriminant_declaration Xd  )
                    ==>(

discriminant_specification   ::=
     Xd  expression
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

entry_call   ::=
     name Xb  ;
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

entry_declaration   ::=
     ENTRY identifier Xep Xeq ;
                    ==>ENTRY

enumeration_literal   ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     character_literal
                    ==>character_literal

enumeration_type_definition   ::=
     ( enumeration_literal Xc  )
                    ==>(

enumeration_type_representation   ::=
     FOR name USE aggregate ;
                    ==>FOR

exception_choice   ::=
     name  |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     OTHERS
                    ==>OTHERS

exception_handler   ::=
     WHEN exception_choice Xb  => sequence_of_statements
                    ==>WHEN

exit_statement   ::=
     EXIT Xa  Xaj ;
                    ==>EXIT

exponent   ::=
     E exponent1
```

```
                    ==>E

exponentiating_operator   ::=
     **
                    ==>**

exponentl   ::=
     Xbp integer  |
                    ==>+ digit
     - integer
                    ==>-

expression   ::=
     relation Xcf  |
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     relation Xcg  |
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     relation Xch  |
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     relation Xci  |
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     relation Xc
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

extended_digit   ::=
     digit  |
                    ==>digit
     letter
                    ==>lower_case_letter upper_case_letter

factor   ::=
     primary Xcj
                    ==>( "{ character }" character_literal digit
                    ==>lower_case_letter typemark upper_case_letter
                    ==>NEW NULL

fixed_point_constraint   ::=
     DELTA simple_expression Xdb
                    ==>DELTA
```

```
floating_point_constraint   ::=
     DIGITS simple_expression Xda
                    ==>DIGITS

formal_parameter   ::=
     identifier
                    ==>lower_case_letter upper_case_letter

formal_part   ::=
     ( parameter_declaration Xdq )
                    ==>(

function_call   ::=
     name actual_parameter_part  |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     name ( )
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

generic_actual_parameter   ::=
     expression  |
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     name  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     subtype indication
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

generic_association   ::=
     Xfh generic_actual_parameter
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

generic_formal_parameter   ::=
     parameter_declaration ;  |
                    ==>lower_case_letter upper_case_letter
     TYPE identifier Xfd IS generic_type_definition ;  |
                    ==>TYPE
     WITH subprogram_specification Xfe ;  |
                    ==>WITH
     WITH subprogram_specification IS "<>" ;
     ***NOT LL(1)***
                    ==>WITH
```

```
generic_package_declaration   ::=
     generic_part package_specification ;
                    ==>GENERIC

generic_package_instantiation   ::=
     NEW name Xff
                    ==>NEW

generic_part   ::=
     GENERIC Xfc
                    ==>GENERIC

generic_subprogram_declaration   ::=
     generic_part subprogram_specification ;
                    ==>GENERIC

generic_subprogram_instantiation   ::=
     PROCEDURE identifier IS generic_instantiation ;  |
                    ==>PROCEDURE
     FUNCTION designator IS generic_instantiation ;
                    ==>FUNCTION

generic_type_definition   ::=
     ( "<>" )  |
                    ==>(
     RANGE "<>"  |
                    ==>RANGE
     DELTA "<>"  |
                    ==>DELTA
     DIGITS "<>"  |
                    ==>DIGITS
     array_type_definition  |
                    ==>ARRAY
     access_type_definition  |
                    ==>ACCESS
     private_type_definition
                    ==>LIMITED PRIVATE

goto_statement   ::=
     GOTO name ;
                    ==>GOTO

identifier   ::=
     letter Xbj
                    ==>lower_case_letter upper_case_letter

identifier_list   ::=
     identifier Xcp
                    ==>lower_case_letter upper_case_letter

if_statement   ::=
     IF condition THEN sequence_of_statements Xae Xaf END IF ;
                    ==>IF
```

```
incomplete_type_declaration   ::=
     TYPE identifier Xdk ;
                  ==>TYPE

index_constraint   ::=
     ( discrete_range Xdf )
                  ==>(

indexed_component   ::=
     name ( expression Xcb )
                  ==>"{ character }" lower_case_letter upper_case_letter
                  ==>

index1   ::=
     type_mark RANGE "<>"  |
                  ==>"{ character }" lower_case_letter upper_case_letter
                  ==>
     type_mark Xde  |
     ***NOT LL(1)***
                  ==>"{ character }" lower_case_letter upper_case_letter
                  ==>
     range
     ***NOT LL(1)***
                  ==>( + - "{ character }" character_literal
                  ==>digit lower_case_letter typemark upper_case_letter
                  ==>NEW NOT NULL

integer   ::=
     digit Xbn
                  ==>digit

integer_type_definition   ::=
     range_constraint
                  ==>RANGE

iteration_clause   ::=
     FOR loop_parameter IN Xa  discrete_range  |
                  ==>FOR
     WHILE condition
                  ==>WHILE

label   ::=
     "<<" identifier ">>"
                  ==>"<<"

length specification   ::=
     FOR attribute USE expression ;
                  ==>FOR

letter   ::=
     upper_case_letter  |
                  ==>upper_case_letter
     lower_case_letter
                    ==>lower_case_letter

letter_or_digit   ::=
     letter  |
                    ==>lower_case_letter upper_case_letter
     digit
                    ==>digit

literal   ::=
     numeric_literal  |
                    ==>digit
     enumeration_literal  |
                    ==>character_literal lower_case_letter upper_case_letter
                    ==>
     character_string  |
                    ==>"{ character }"
     NULL
                    ==>NULL

location   ::=
     AT simple_expression RANGE range
                    ==>AT

logical_operator   ::=
     AND  |
                    ==>AND
     OR  |
                    ==>OR
     XOR
                    ==>XOR

loop_parameter   ::=
     identifier
                    ==>lower_case_letter upper_case_letter

loop_statement   ::=
     Xai basic_loop Xa  ;
                    ==>FOR LOOP WHILE

mode   ::=
     Xd   |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>IN
     OUT  |
                    ==>OUT
     IN OUT
     ***NOT LL(1)***
                    ==>IN

multiplying_operator   ::=
     *  |
                    ==>*
     /  |
                    ==>/
     MOD  |
                    ==>MOD
     REM
                    ==>REM

name   ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     indexed_component  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     slice  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     selected_component  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     attribute  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     function_call  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     operator_symbol
     ***NOT LL(1)***
                    ==>"{ character }"

null_statement   ::=
     NULL ;
                    ==>NULL

numeric_literal   ::=
     decimal_number  |
                    ==>digit
     based_number
     ***NOT LL(1)***
                    ==>digit

object_or_number_declaration   ::=
     Xcl subtype_indication Xcm ;  |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>CONSTANT
     Xcn array_type_definition Xco ;  |
     ***NOT LL(1)***
                    ==>ARRAY CONSTANT
     CONSTANT ":=" expression ;
     ***NOT LL(1)***
                    ==>CONSTANT

operator_symbol   ::=
     character_string
                    ==>"{ character }"

package_body   ::=
     PACKAGE BODY identifier IS declarative_part Xei END Xe  ;
                    ==>PACKAGE

package_decl   ::=
     package_specification  |
                    ==>lower_case_letter upper_case_letter END
                    ==>FUNCTION GENERIC PACKAGE PRIVATE PROCEDURE
                    ==>SUBTYPE TASK TYPE USE
     generic_package_instantiation
                    ==>NEW

package_decl_header   ::=
     PACKAGE identifier IS
                    ==>PACKAGE

package_declaration   ::=
     generic_package_declaration  |
                    ==>GENERIC
     package_decl_header package_decl
                    ==>PACKAGE

package_specification   ::=
     Xed Xee END Xeh
                    ==>lower_case_letter upper_case_letter END
                    ==>FUNCTION GENERIC PACKAGE PRIVATE PROCEDURE
                    ==>SUBTYPE TASK TYPE USE

parameter_association   ::=
     Xb  actual_parameter
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

parameter_declaration   ::=
     identifier_list ":" mode subtype_indication Xdr
                    ==>lower_case_letter upper_case_letter

pragma   ::=
     PRAGMA identifier Xdl ;
                    ==>PRAGMA

primary   ::=
     literal  |
                    ==>"{ character }" character_literal digit
                    ==>lower_case_letter upper_case_letter NULL
                    ==>
     aggregate  |
                    ==>(
     name  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     allocator  |
                    ==>NEW
     function_call  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     type_conversion  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     qualified_expression  |
                    ==>typemark
     ( expression )
     ***NOT LL(1)***
                    ==>(

private_type_definition   ::=
     Xe  PRIVATE
                    ==>LIMITED PRIVATE

procedure_call   ::=
     name Xb  ;
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

program_component   ::=
     body  |
                    ==>FUNCTION PACKAGE PROCEDURE TASK
     package_declaration  |
     ***NOT LL(1)***
                    ==>GENERIC PACKAGE
     task declaration  |
     ***NOT LL(1)***
                    ==>TASK
     body_stub
     ***NOT LL(1)***
                    ==>FUNCTION PACKAGE PROCEDURE TASK

qualified_expression   ::=
     typemark qualified_ex1
                    ==>typemark

qualified_ex1   ::=
     ( expression )  |
                    ==>(
     "'" aggregate
```

```
                    ==>"'"

raise_statement   ::=
     RAISE Xbi ;
                    ==>RAISE

range   ::=
     simple_expression .. simple_expression
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

range_constraint   ::=
     RANGE range
                    ==>RANGE

real_type_definition   ::=
     accuracy_constraint
                    ==>DELTA DIGITS

record_type_definition   ::=
     RECORD component_list END RECORD
                    ==>RECORD

record_type_representation   ::=
     FOR name USE RECORD Xfi Xf  END RECORD ;
                    ==>FOR

relation   ::=
     simple_expression Xc   |
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     simple_expression Xc  IN range  |
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL
     simple_expression Xc  IN subtype_indication
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

relational_operator   ::=
     =  |
                    ==>=
     /=  |
                    ==>/=
     "<"  |
                    ==>"<"
     "<="  |
                    ==>"<="
     ">"  |
                      ==>">"
     ">="
                      ==>">="

renaming_declaration   ::=
     renaming_declaration1 RENAMES name ;  |
                      ==>"{ character }" lower_case_letter upper_case_letter
                      ==>EXCEPTION
     PACKAGE identifier RENAMES name ;  |
                      ==>PACKAGE
     TASK identifier RENAMES name ;  |
                      ==>TASK
     subprogram_specification RENAMES name ;
                      ==>FUNCTION PROCEDURE

renaming_declaration1   ::=
     type_mark  |
                      ==>"{ character }" lower_case_letter upper_case_letter
                      ==>
     EXCEPTION
                      ==>EXCEPTION

representation_specification   ::=
     length_specification  |
                      ==>FOR
     enumeration_type_representation  |
     ***NOT LL(1)***
                      ==>FOR
     record_type_representation  |
     ***NOT LL(1)***
                      ==>FOR
     address_specification
     ***NOT LL(1)***
                      ==>FOR

return_statement   ::=
     RETURN Xak ;
                      ==>RETURN

select_alternative   ::=
     accept_statement Xbc  |
                      ==>ACCEPT
     delay_statement Xbd  |
                      ==>DELAY
     TERMINATE ;
                      ==>TERMINATE

select_statement   ::=
     selective_wait  |
                      ==>SELECT
     conditional_entry_call  |
     ***NOT LL(1)***
```

```
                    ==>SELECT
     timed_entry_call
     ***NOT LL(1)***
                    ==>SELECT

selected_component   ::=
     name . selected_compl
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

selected_compl   ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     ALL  |
                    ==>ALL
     operator_symbol
                    ==>"{ character }"

selective_wait   ::=
     SELECT Xar select_alternative Xa  Xbb END SELECT ;
                    ==>SELECT

sequence_of_statements   ::=
     statement Xaa
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                    ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                    ==>

simple_expression   ::=
     Xc  term Xc
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

simple_statement   ::=
     null_statement  |
                    ==>NULL
     assignment_statement  |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     exit_statement  |
                    ==>EXIT
     return_statement  |
                    ==>RETURN
     goto_statement  |
                    ==>GOTO
     procedure_call  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     entry_call  |
     ***NOT LL(1)***
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     delay_statement  |
                    ==>DELAY
     abort_statement  |
                    ==>ABORT
     raise_statement  |
                    ==>RAISE
     code_statement
                    ==>typemark

slice   ::=
     name ( discrete_range )
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

statement   ::=
     Xab simple_statement  |
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT DELAY
                    ==>EXIT GOTO NULL RAISE RETURN
     Xac compound_statement
     ***NOT LL(1)***
                    ==>"<<" lower_case_letter upper_case_letter
                    ==>ACCEPT BEGIN CASE DECLARE FOR IF LOOP
                    ==>SELECT WHILE

subprogram_body   ::=
     subprogram_specification IS declarative_part BEGIN sequence_of_statements
          Xea END Xec ;
                    ==>FUNCTION PROCEDURE

subprogram_declaration   ::=
     subprogram_specification ;  |
                    ==>FUNCTION PROCEDURE
     generic_subprogram_declaration  |
                    ==>GENERIC
     generic_subprogram_instantiation
     ***NOT LL(1)***
                    ==>FUNCTION PROCEDURE

subprogram_specification   ::=
     PROCEDURE identifier Xdo  |
                    ==>PROCEDURE
     FUNCTION designator Xdp RETURN subtype_indication
                    ==>FUNCTION

subtype_declaration   ::=
     SUBTYPE identifier IS subtype_indication
                    ==>SUBTYPE

subtype_indication   ::=
     type_mark Xcr
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

subunit   ::=
     SEPARATE ( name ) subunit_body
                    ==>SEPARATE

subunit_body   ::=
     subprogram_body  |
                    ==>FUNCTION PROCEDURE
     package_body  |
                    ==>PACKAGE
     task_body
                    ==>TASK

task_body   ::=
     TASK BODY identifier IS Xel BEGIN sequence_of_statements Xem END Xeo ;
                    ==>TASK

task_declaration   ::=
     task_specification
                    ==>TASK

task_specification   ::=
     TASK Xe  identifier Xe  ;
                    ==>TASK

term   ::=
     factor Xc
                    ==>( "{ character }" character_literal digit
                    ==>lower_case_letter typemark upper_case_letter
                    ==>NEW NULL

timed_entry_call   ::=
     SELECT entry_call Xbf OR delay_statement Xbg END SELECT ;
                    ==>SELECT

type_conversion   ::=
     type_mark ( expression )
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

type_declaration   ::=
     TYPE identifier Xcq IS type_definition ;  |
                    ==>TYPE
     incomplete_type_declaration
     ***NOT LL(1)***
                    ==>TYPE

type_definition   ::=
     enumeration_type_definition  |
                    ==>(
     integer_type_definition  |
                    ==>RANGE
     real_type_definition  |
                    ==>DELTA DIGITS
     array_type_definition  |
                    ==>ARRAY
     record_type_definition  |
                    ==>RECORD
     access_type_definition  |
                    ==>ACCESS
     derived_type_definition  |
                    ==>NEW
     private_type_definition
                    ==>LIMITED PRIVATE

type_mark   ::=
     name
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

unary_operator   ::=
     +  |
                    ==>+
     -  |
                    ==>-
     NOT
                    ==>NOT

use_clause   ::=
     USE name Xb  ;
                    ==>USE

variant_part   ::=
     CASE name IS Xd  END CASE ;
                    ==>CASE

with_clause   ::=
     WITH name Xfb ;
                    ==>WITH

Xaa   ::=
     statement Xaa  |
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                    ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                    ==>
     <empty>
     ***NOT LL(1)***
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY ELSE ELSIF END
                    ==>EXCEPTION EXIT FOR GOTO IF LOOP NULL
                    ==>OR RAISE RETURN SELECT WHEN WHILE

Xab   ::=
     label Xab  |
                    ==>"<<"
     <empty>
                    ==>"{ character }" lower_case_letter typemark
                    ==>upper_case_letter ABORT DELAY EXIT GOTO
                    ==>NULL RAISE RETURN

Xac   ::=
     label Xac  |
                    ==>"<<"
     <empty>
                    ==>lower_case_letter upper_case_letter ACCEPT
                    ==>BEGIN CASE DECLARE FOR IF LOOP SELECT
                    ==>WHILE

Xad   ::=
     identifier ":"  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>BEGIN DECLARE FOR LOOP WHILE

Xae   ::=
     ELSIF condition THEN sequence_of_statements Xae  |
                    ==>ELSIF
     <empty>
                    ==>ELSE END

Xaf   ::=
     ELSE sequence_of_statements  |
                    ==>ELSE
     <empty>
                    ==>END

Xag   ::=
     WHEN choice Xah => sequence_of_statements Xag  |
                    ==>WHEN
     <empty>
     ***NOT LL(1)***
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY ELSE ELSIF END
                    ==>EXCEPTION EXIT FOR GOTO IF LOOP NULL
                    ==>OR RAISE RETURN SELECT WHEN WHILE

Xah   ::=
     "|" choice Xah  |
                    ==>"|"
     <empty>
                    ==>=>
```

```
Xai   ::=
     iteration_clause  |
                    ==>FOR WHILE
     <empty>
                    ==>LOOP

Xa    ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>;

Xa    ::=
     REVERSE  |
                    ==>REVERSE
     <empty>
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

Xa    ::=
     DECLARE declarative_part  |
                    ==>DECLARE
     <empty>
                    ==>BEGIN

Xa    ::=
     EXCEPTION Xa   |
                    ==>EXCEPTION
     <empty>
                    ==>END

Xa    ::=
     exception_handler Xa   |
                    ==>WHEN
     <empty>
                    ==>END

Xa    ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>;

Xa    ::=
     name  |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     <empty>
                    ==>; WHEN

Xaj   ::=
     WHEN condition  |
                       ==>WHEN
    <empty>
                       ==>;

Xak   ::=
    expression  |
                       ==>( + - "{ character }" character_literal
                       ==>digit lower_case_letter typemark upper_case_letter
                       ==>NEW NOT NULL
    <empty>
                       ==>;

Xal   ::=
    declarative_item Xal  |
                       ==>lower_case_letter upper_case_letter FUNCTION
                       ==>GENERIC PACKAGE PROCEDURE SUBTYPE TASK
                       ==>TYPE USE
    <empty>
    ***NOT LL(1)***
                       ==>BEGIN END FOR FUNCTION GENERIC PACKAGE
                       ==>PROCEDURE TASK

Xam   ::=
    representation_specification Xam  |
                       ==>FOR
    <empty>
                       ==>BEGIN END FUNCTION GENERIC PACKAGE PROCEDURE
                       ==>TASK

Xan   ::=
    program_component Xan  |
                       ==>FUNCTION GENERIC PACKAGE PROCEDURE TASK
                       ==>
    <empty>
                       ==>BEGIN END

Xao   ::=
    formal_part  |
                       ==>(
    <empty>
                       ==>; DO

Xap   ::=
    DO sequence_of_statements END Xaq  |
                       ==>DO
    <empty>
                       ==>;

Xaq   ::=
    identifier  |
                       ==>lower_case_letter upper_case_letter
    <empty>
                       ==>;
```

```
Xar    ::=
     WHEN condition =>  |
                    ==>WHEN
     <empty>
                    ==>ACCEPT DELAY TERMINATE

Xa     ::=
     OR Xba select_alternative Xa   |
                    ==>OR
     <empty>
                    ==>ELSE END

Xba    ::=
     WHEN condition =>  |
                    ==>WHEN
     <empty>
                    ==>ACCEPT DELAY TERMINATE

Xbb    ::=
     ELSE sequence_of_statements  |
                    ==>ELSE
     <empty>
                    ==>END

Xbc    ::=
     sequence_of_statements  |
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                    ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                    ==>
     <empty>
                    ==>ELSE END OR

Xbd    ::=
     sequence_of_statements  |
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                    ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                    ==>
     <empty>
                    ==>ELSE END OR

Xbe    ::=
     sequence_of_statements  |
                    ==>"<<" "{ character }" lower_case_letter
                    ==>typemark upper_case_letter ABORT ACCEPT
                    ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                    ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                    ==>
     <empty>
```

```
                     ==>ELSE

Xbf   ::=
     sequence_of_statements  |
                     ==>"<<" "{ character }" lower_case_letter
                     ==>typemark upper_case_letter ABORT ACCEPT
                     ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                     ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                     ==>
     <empty>
                     ==>OR

Xbg   ::=
     sequence_of_statements  |
                     ==>"<<" "{ character }" lower_case_letter
                     ==>typemark upper_case_letter ABORT ACCEPT
                     ==>BEGIN CASE DECLARE DELAY EXIT FOR GOTO
                     ==>IF LOOP NULL RAISE RETURN SELECT WHILE
                     ==>
     <empty>
                     [illegible]

Xbh   ::=
     , name Xbh  |
                     ==>,
     <empty>
                     ==>;

Xbi   ::=
     name  |
                     ==>"{ character }" lower_case_letter upper_case_letter
                     ==>
     <empty>
                     ==>;

Xb    ::=
     range_constraint  |
                     ==>RANGE
     <empty>
                     ==>) , => "|" LOOP

Xb    ::=
     , name Xb   |
                     ==>,
     <empty>
                     ==>;

Xb    ::=
     , parameter_association Xb   |
                     ==>,
     <empty>
                     ==>)
```

```
Xb    ::=
     formal_parameter =>  |
                    ==>lower_case_letter upper_case_letter
     <empty>
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

Xb    ::=
     actual_parameter_part  |
                    ==>(
     <empty>
                    ==>;

Xb    ::=
     "|" exception_choice Xb   |
                    ==>"|"
     <empty>
                    ==>=>

Xb    ::=
     ( actual_parameter_part )  |
                    ==>(
     <empty>
                    ==>;

Xbj   ::=
     Xbk letter_or_digit Xbj  |
                    ==>digit lower_case_letter underscore upper_case_letter
                    ==>
     <empty>
     ***NOT LL(1)***
                    ==>. .. ( + & * ** ) ; - / /= , = => "<="
                    ==>"<" "|" ">>" ">=" ">" ":=" ":" "'" lower_case_letter
                    ==>upper_case_letter AND ARRAY AT BEGIN
                    ==>DELTA DIGITS DO END FOR FUNCTION GENERIC
                    ==>IN IS LOOP MOD NOT OR PACKAGE PRIVATE
                    ==>PROCEDURE RANGE REM RENAMES RETURN SEPARATE
                    ==>SUBTYPE TASK THEN TYPE USE WHEN WITH
                    ==>XOR

Xbk   ::=
     underscore  |
                    ==>underscore
     <empty>
                    ==>digit lower_case_letter upper_case_letter
                    ==>

Xbl   ::=
     . integer  |
                    ==>.
     <empty>
```

```
                          ==>.. + & * ** ) ; - / /= , = => "<=" "<"
                          ==>"|" ">=" ">" ":=" lower_case_letter upper_case_letter
                          ==>AND ARRAY BEGIN E END FOR FUNCTION GENERIC
                          ==>IN IS LOOP MOD NOT OR PACKAGE PRIVATE
                          ==>PROCEDURE RANGE REM RENAMES SUBTYPE TASK
                          ==>THEN TYPE USE XOR

Xbm   ::=
     exponent  |
                          ==>E
     <empty>
                          ==>.. + & * ** ) ; - / /= , = => "<=" "<"
                          ==>"|" ">=" ">" ":=" lower_case_letter upper_case_letter
                          ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                          ==>IN IS LOOP MOD NOT OR PACKAGE PRIVATE
                          ==>PROCEDURE RANGE REM RENAMES SUBTYPE TASK
                          ==>THEN TYPE USE XOR

Xbn   ::=
     Xbo digit Xbn  |
                          ==>digit underscore
     <empty>
                          ==>. .. + & * ** ) ; - / /= , # = => "<="
                          ==>"<" "|" ">=" ">" ":=" lower_case_letter
                          ==>upper_case_letter AND ARRAY BEGIN E END
                          ==>FOR FUNCTION GENERIC IN IS LOOP MOD NOT
                          ==>OR PACKAGE PRIVATE PROCEDURE RANGE REM
                          ==>RENAMES SUBTYPE TASK THEN TYPE USE XOR
                          ==>

Xbo   ::=
     underscore  |
                          ==>underscore
     <empty>
                          ==>digit

Xbp   ::=
     +  |
                          ==>+
     <empty>
                          ==>digit

Xbq   ::=
     . based_integer  |
                          ==>.
     <empty>
                          ==>#

Xbr   ::=
     exponent  |
                          ==>E
     <empty>
                          ==>.. + & * ** ) ; - / /= , = => "<=" "<"
```

```
                    ==>"|" ">=" ">" ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IN IS LOOP MOD NOT OR PACKAGE PRIVATE
                    ==>PROCEDURE RANGE REM RENAMES SUBTYPE TASK
                    ==>THEN TYPE USE XOR

Xb    ::=
     Xca extended_digit Xb   |
                    ==>digit lower_case_letter underscore upper_case_letter
                    ==>
     <empty>
                    ==>. #

Xca   ::=
     underscore  |
                    ==>underscore
     <empty>
                    ==>digit lower_case_letter upper_case_letter
                    ==>

Xcb   ::=
     , expression Xcb  |
                    ==>,
     <empty>
                    ==>)

Xcc   ::=
     , component_association Xcc  |
                    ==>,
     <empty>
                    ==>)

Xcd   ::=
     choice Xce =>  |
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL OTHERS
     <empty>
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

Xce   ::=
     "|" choice Xce  |
                    ==>"|"
     <empty>
                    ==>=>

Xcf   ::=
     AND relation Xcf  |
                    ==>AND
     <empty>
```

```
                              ==>) ; , => IS LOOP THEN

Xcg   ::=
     OR relation Xcg  |
                              ==>OR
     <empty>
                              ==>) ; , => IS LOOP THEN

Xch   ::=
     XOR relation Xch  |
                              ==>XOR
     <empty>
                              ==>) ; , => IS LOOP THEN

Xci   ::=
     AND THEN relation Xci  |
                              ==>AND
     <empty>
                              ==>) ; , => IS LOOP THEN

Xc    ::=
     OR ELSE relation Xc   |
                              ==>OR
     <empty>
                              ==>) ; , => IS LOOP THEN

Xc    ::=
     relational_operator simple_expression  |
                              ==>/= = "<=" "<" ">=" ">"
     <empty>
                              ==>) ; , => AND IS LOOP OR THEN XOR

Xc    ::=
     NOT  |
                              ==>NOT
     <empty>
                              ==>IN

Xc    ::=
     NOT  |
                              ==>NOT
     <empty>
                              ==>IN

Xc    ::=
     unary_operator  |
                              ==>+ - NOT
     <empty>
                              ==>( "{ character }" character_literal digit
                              ==>lower_case_letter typemark upper_case_letter
                              ==>NEW NULL

Xc    ::=
     adding_operator term Xc    |
                    ==>+ & -
     <empty>
                    ==>.. ) ; /= , = => "<=" "<" "|" ">=" ">"
                    ==>":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IN IS LOOP NOT OR PACKAGE PRIVATE PROCEDURE
                    ==>RANGE RENAMES SUBTYPE TASK THEN TYPE
                    ==>USE XOR

Xc    ::=
     multiplying_operator factor Xc    |
                    ==>* / MOD REM
     <empty>
                    ==>.. + & ) ; - /= , = => "<=" "<" "|" ">="
                    ==>">" ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IN IS LOOP NOT OR PACKAGE PRIVATE PROCEDURE
                    ==>RANGE RENAMES SUBTYPE TASK THEN TYPE
                    ==>USE XOR

Xcj   ::=
     ** primary   |
                    ==>**
     <empty>
                    ==>.. + & * ) ; - / /= , = => "<=" "<" "|"
                    ==>">=" ">" ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IN IS LOOP MOD NOT OR PACKAGE PRIVATE
                    ==>PROCEDURE RANGE REM RENAMES SUBTYPE TASK
                    ==>THEN TYPE USE XOR

Xck   ::=
     ( expression )  |
                    ==>(
     <empty>
                    ==>.. + & * ** ) ; - / /= , = => "<=" "<"
                    ==>"|" ">=" ">" ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IN IS LOOP MOD NOT OR PACKAGE PRIVATE
                    ==>PROCEDURE RANGE REM RENAMES SUBTYPE TASK
                    ==>THEN TYPE USE XOR

Xcl   ::=
     CONSTANT   |
                    ==>CONSTANT
     <empty>
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>

Xcm   ::=
     ":=" expression  |
                    ==>":="
     <empty>
                    ==>;

Xcn   ::=
     CONSTANT  |
                    ==>CONSTANT
     <empty>
                    ==>ARRAY

Xco   ::=
     ":=" expression  |
                    ==>":="
     <empty>
                    ==>;

Xcp   ::=
     , identifier Xcp  |
                    ==>,
     <empty>
                    ==>":"

Xcq   ::=
     discriminant_part  |
                    ==>(
     <empty>
                    ==>IS

Xcr   ::=
     constraint  |
                    ==>( DELTA DIGITS RANGE
     <empty>
                    ==>) ; , => ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IS LOOP OR PACKAGE PRIVATE PROCEDURE
                    ==>RENAMES SUBTYPE TASK THEN TYPE USE XOR
                    ==>

Xc    ::=
     , enumeration_literal Xc   |
                    ==>,
     <empty>
                    ==>)

Xda   ::=
     range_constraint  |
                    ==>RANGE
     <empty>
                    ==>) ; , => ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IS LOOP OR PACKAGE PRIVATE PROCEDURE
                    ==>RENAMES SUBTYPE TASK THEN TYPE USE XOR
                    ==>
```

```
Xdb   ::=
     range_constraint  |
                    ==>RANGE
     <empty>
                    ==>) ; , => ":=" lower_case_letter upper_case_letter
                    ==>AND ARRAY BEGIN END FOR FUNCTION GENERIC
                    ==>IS LOOP OR PACKAGE PRIVATE PROCEDURE
                    ==>RENAMES SUBTYPE TASK THEN TYPE USE XOR
                    ==>

Xdc   ::=
     , index Xdc  |
                    ==>,
     <empty>
                    ==>)

Xdd   ::=
     , index1 Xdd  |
                    ==>,
     <empty>
                    ==>)

Xde   ::=
     range_constraint  |
                    ==>RANGE
     <empty>
                    ==>) ,

Xdf   ::=
     , discrete_range Xdf  |
                    ==>,
     <empty>
                    ==>)

Xdg   ::=
     component_declaration Xdg  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>CASE END WHEN

Xdh   ::=
     variant_part  |
                    ==>CASE
     <empty>
                    ==>END WHEN

Xdi   ::=
     ":=" expression  |
                    ==>":="
     <empty>
                    ==>;

Xd    ::=
     ":=" expression  |
                  ==>":="
     <empty>
                  ==>;

Xd    ::=
     ; discriminant_declaration Xd   |
                  ==>;
     <empty>
                  ==>)

Xd    ::=
     ":=" expression  |
                  ==>":="
     <empty>
                  ==>) ;

Xd    ::=
     , discriminant_specification Xd   |
                  ==>,
     <empty>
                  ==>)

Xd    ::=
     name Xd  =>  |
                  ==>"{ character }" lower_case_letter upper_case_letter
                  ==>
     <empty>
     ***NOT LL(1)***
                  ==>( + - "{ character }" character_literal
                  ==>digit lower_case_letter typemark upper_case_letter
                  ==>NEW NOT NULL

Xd    ::=
     "|" name Xd   |
                  ==>"|"
     <empty>
                  ==>=>

Xd    ::=
     WHEN choice Xdj => component_list Xd   |
                  ==>WHEN
     <empty>
                  ==>END

Xdj   ::=
     "|" choice Xdj  |
                  ==>"|"
     <empty>
                  ==>=>

Xdk   ::=
     discriminant_part  |
                         ==>(
     <empty>
                         ==>;

Xdl    ::=
     ( argument Xdm )  |
                         ==>(
     <empty>
                         ==>;

Xdm    ::=
     , argument Xdm  |
                         ==>,
     <empty>
                         ==>)

Xdn    ::=
     identifier =>  |
                         ==>lower_case_letter upper_case_letter
     <empty>
     ***NOT LL(1)***
                         ==>( + - "{ character }" character_literal
                         ==>digit lower_case_letter typemark upper_case_letter
                         ==>NEW NOT NULL

Xdo    ::=
     formal_part  |
                         ==>(
     <empty>
                         ==>; IS RENAMES

Xdp    ::=
     formal_part  |
                         ==>(
     <empty>
                         ==>RETURN

Xdq    ::=
     ; parameter_declaration Xdq  |
                         ==>;
     <empty>
                         ==>)

Xdr    ::=
     ":=" expression  |
                         ==>":="
     <empty>
                         ==>) ;

Xd     ::=
     IN  |
                         ==>IN
     <empty>
```

```
                     ==>"{ character }" lower_case_letter upper_case_letter
                     ==>

Xea    ::=
       EXCEPTION Xeb  |
                     ==>EXCEPTION
       <empty>
                     ==>END

Xeb    ::=
       exception_handler Xeb  |
                     ==>WHEN
       <empty>
                     ==>END

Xec    ::=
       designator  |
                     ==>"{ character }" lower_case_letter upper_case_letter
                     ==>
       <empty>
                     ==>;

Xed    ::=
       declarative_item Xed  |
                     ==>lower_case_letter upper_case_letter FUNCTION
                     ==>GENERIC PACKAGE PROCEDURE SUBTYPE TASK
                     ==>TYPE USE
       <empty>
                     ==>END PRIVATE

Xee    ::=
       PRIVATE Xef Xeg  |
                     ==>PRIVATE
       <empty>
                     ==>END

Xef    ::=
       declarative_item Xef  |
                     ==>lower_case_letter upper_case_letter FUNCTION
                     ==>GENERIC PACKAGE PROCEDURE SUBTYPE TASK
                     ==>TYPE USE
       <empty>
                     ==>END FOR

Xeg    ::=
       representation_specification Xeg  |
                     ==>FOR
       <empty>
                     ==>END

Xeh    ::=
       identifier  |
                     ==>lower_case_letter upper_case_letter
```

```
     <empty>
     ***NOT LL(1)***
                    ==>; lower_case_letter upper_case_letter
                    ==>BEGIN END FOR FUNCTION GENERIC PACKAGE
                    ==>PRIVATE PROCEDURE SEPARATE SUBTYPE TASK
                    ==>TYPE USE WITH

Xei    ::=
     BEGIN sequence_of_statements Xe    |
                    ==>BEGIN
     <empty>
                    ==>END

Xe     ::=
     EXCEPTION Xe   |
                    ==>EXCEPTION
     <empty>
                    ==>END

Xe     ::=
     exception_handler Xe   |
                    ==>WHEN
     <empty>
                    ==>END

Xe     ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>;

Xe     ::=
     LIMITED  |
                    ==>LIMITED
     <empty>
                    ==>PRIVATE

Xe     ::=
     TYPE  |
                    ==>TYPE
     <empty>
                    ==>lower_case_letter upper_case_letter

Xe     ::=
     IS Xe  Xej END Xek  |
                    ==>IS
     <empty>
                    ==>;

Xe     ::=
     entry_declaration Xe   |
                    ==>ENTRY
     <empty>
```

```
                    ==>END FOR

Xej   ::=
     representation_specification Xej  |
                    ==>FOR
     <empty>
                    ==>END

Xek   ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>;

Xel   ::=
     declarative_part  |
                    ==>lower_case_letter upper_case_letter BEGIN
                    ==>FOR FUNCTION GENERIC PACKAGE PROCEDURE
                    ==>SUBTYPE TASK TYPE USE
     <empty>
     ***NOT LL(1)***
                    ==>BEGIN

Xem   ::=
     EXCEPTION Xen  |
                    ==>EXCEPTION
     <empty>
                    ==>END

Xen   ::=
     exception_handler Xen  |
                    ==>WHEN
     <empty>
                    ==>END

Xeo   ::=
     identifier  |
                    ==>lower_case_letter upper_case_letter
     <empty>
                    ==>;

Xep   ::=
     ( discrete_range )  |
                    ==>(
     <empty>
     ***NOT LL(1)***
                    ==>( ;

Xeq   ::=
     formal_part  |
                    ==>(
     <empty>
                    ==>;
```

```
Xer    ::=
     compilation_unit Xer  |
                    ==>FUNCTION GENERIC PACKAGE PROCEDURE SEPARATE
                    ==>WITH
     <empty>
                    ==>

Xe     ::=
     with_clause Xfa Xe   |
                    ==>WITH
     <empty>
                    ==>FUNCTION GENERIC PACKAGE PROCEDURE SEPARATE
                    ==>

Xfa    ::=
     use_clause  |
                    ==>USE
     <empty>
                    ==>FUNCTION GENERIC PACKAGE PROCEDURE SEPARATE
                    ==>WITH

Xfb    ::=
     , name Xfb  |
                    ==>,
     <empty>
                    ==>;

Xfc    ::=
     generic_formal_parameter Xfc  |
                    ==>lower_case_letter upper_case_letter TYPE
                    ==>WITH
     <empty>
     ***NOT LL(1)***
                    ==>lower_case_letter upper_case_letter END
                    ==>FUNCTION GENERIC PACKAGE PRIVATE PROCEDURE
                    ==>SUBTYPE TASK TYPE USE

Xfd    ::=
     discriminant_part  |
                    ==>(
     <empty>
                    ==>IS

Xfe    ::=
     IS name  |
                    ==>IS
     <empty>
                    ==>;

Xff    ::=
     ( generic_association Xfg )  |
                    ==>(
     <empty>
                    ==>lower_case_letter upper_case_letter BEGIN
                    ==>END FOR FUNCTION GENERIC PACKAGE PRIVATE
                    ==>PROCEDURE SEPARATE SUBTYPE TASK TYPE
                    ==>USE WITH

Xfg    ::=
     , generic_association Xfg  |
                    ==>,
     <empty>
                    ==>)

Xfh    ::=
     formal_parameter =>  |
                    ==>lower_case_letter upper_case_letter
     <empty>
     ***NOT LL(1)***
                    ==>( + - "{ character }" character_literal
                    ==>digit lower_case_letter typemark upper_case_letter
                    ==>NEW NOT NULL

Xfi    ::=
     alignment_clause ;  |
                    ==>AT
     <empty>
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>END

Xf     ::=
     name location ; Xf   |
                    ==>"{ character }" lower_case_letter upper_case_letter
                    ==>
     <empty>
                    ==>END

***GRAMMAR IS NOT LL(1)***
```

APPENDIX D

THE SYMBOL DICTIONARY

D.1 Underlying Data Structures

The data structures which represent the current compilation context are the compiler's source of information fro associating symbols with their attributes. A symbol can be an identifier, an operator, a literal, predefined attributes and pragmas, and reserved words. The structures which define the compilation context are the Visibility Stack, the declared symbol tables, the Look-Up Table and the Name Table. The composition of these data structures constitutes the symbol dictionary (cf. Figure D-1).

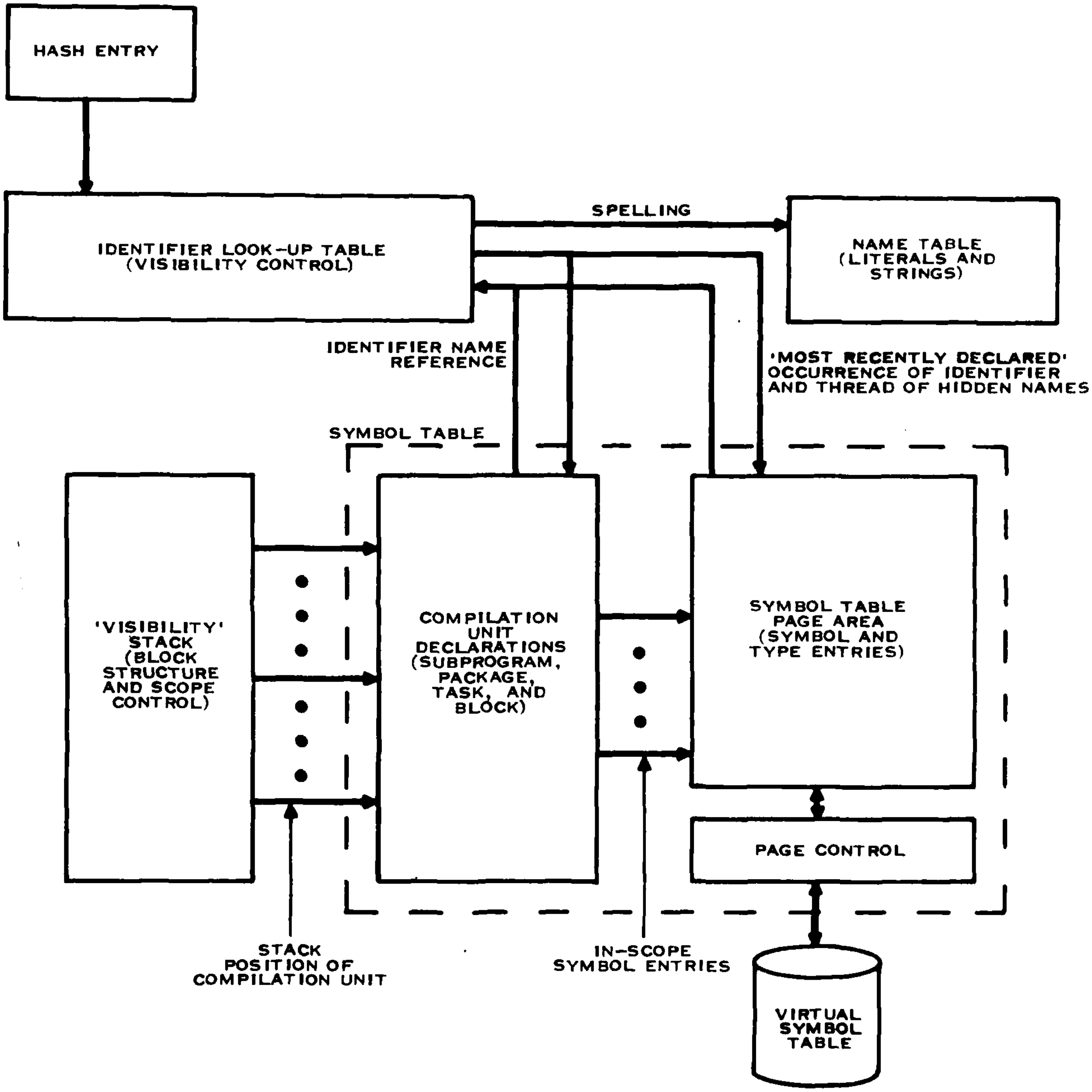

HASH ENTRY
IDENTIFIER LOOK-UP TABLE (VISIBILITY CONTROL)
SPELLING
NAME TABLE (LITERALS AND STRINGS)
IDENTIFIER NAME REFERENCE
'MOST RECENTLY DECLARED' OCCURRENCE OF IDENTIFIER AND THREAD OF HIDDEN NAMES
SYMBOL TABLE
'VISIBILITY' STACK (BLOCK STRUCTURE AND SCOPE CONTROL)
COMPILATION UNIT DECLARATIONS (SUBPROGRAM, PACKAGE, TASK, AND BLOCK)
SYMBOL TABLE PAGE AREA (SYMBOL AND TYPE ENTRIES)
PAGE CONTROL
STACK POSITION OF COMPILATION UNIT
IN-SCOPE SYMBOL ENTRIES
VIRTUAL SYMBOL TABLE

Figure D-1 Organization of the Symbol Dictionary

D.1.1 The Visibility Stack

A compile-time Visibility Stack (cf. Figure D-2) is used to effect the visibility rules and maintain the block structure. It is expandable in both directions from the entry for the package STANDARD in order to handle WITHed packages and subprograms.

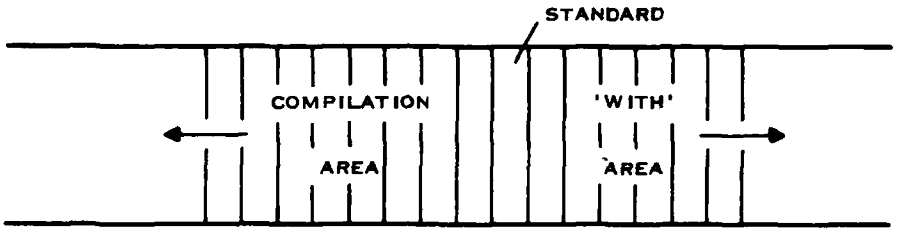


Figure D-2 Structure of the Visibility Stack

An element of the compile-time Visibility Stack entry contains (cf. Figure D-3):

1. A reference to the compilation unit/sub-unit specification. This reference to a subprogram, package or block declarator provides callability and visibility to the specification and visibility to its symbol table. Relative position in the Visibility Stack provides information regarding the hierarchical relation of the unit with the compilation context and its callability. Accessability to the specification and its symbol table is gained through direct visibility or selected component reference through the name of the declared block. The stack allows the reference of a specification for a procedure/function call, if it is in scope, even though its symbols are not in scope. In this circumstance, the pointer from the unit block declarator (cf. Section D.1.2.1) to the symbol table would have been removed and set to NIL.

2. The head of a 'Use' chain which links packages used since the start of the unit's declaration.

3. A reference to the Visibility Stack which encompasses subunits declared within the scope of the parent unit.

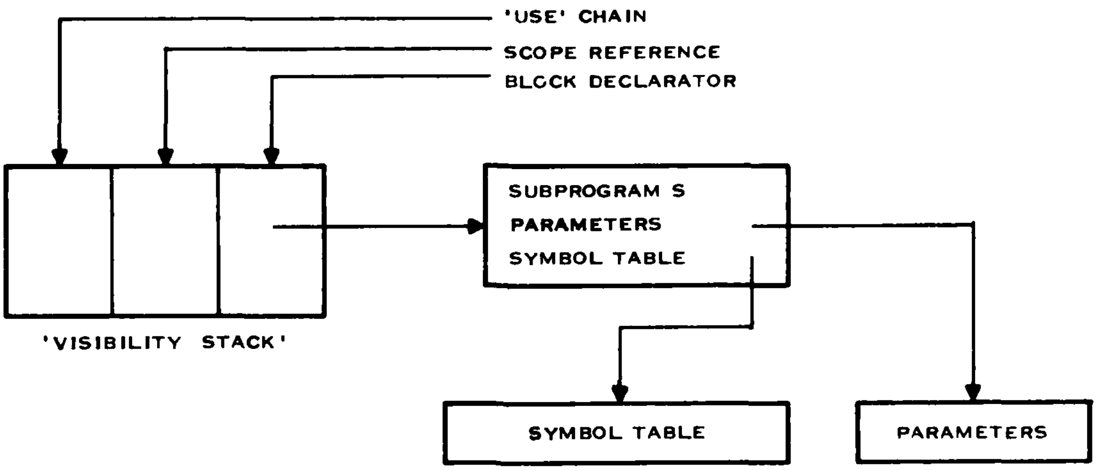


Figure D-3 Visibility Stack Entry

D.1.2 Declared Symbol Tables

There is a separate symbol table for each language construct that can contain a declarative part (i.e., block, subprogram, package or task). Symbol tables are referenced from the Visibility Stack to form the compilation context for a block, subprogram, package or task (cf. Figure D-4).

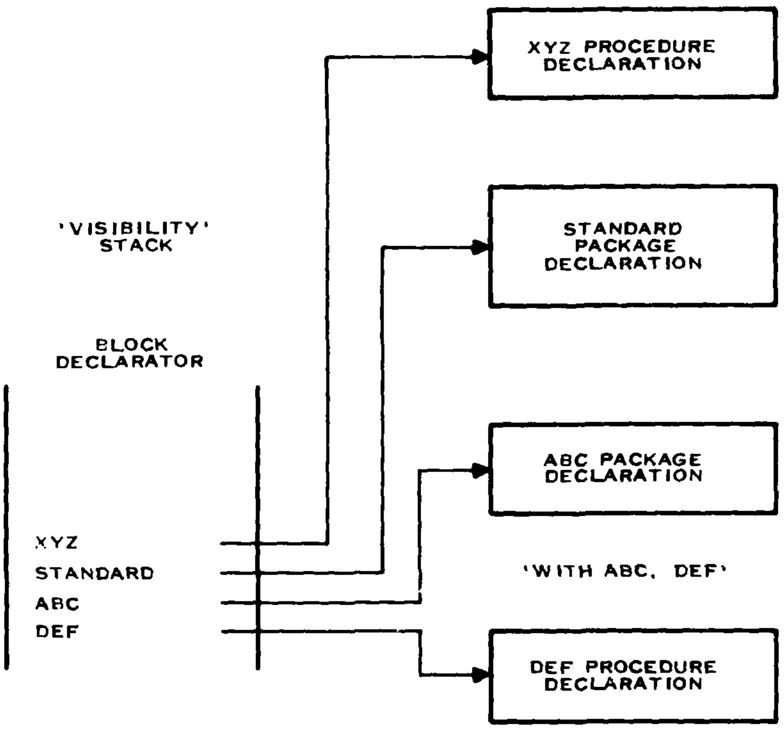


Figure D-4 Organization of Declared Symbol Tables

The entries within the symbol tables are interconnected together in a manner which realizes the semantics of the visibility rules (i.e., scope, hiding, overloading, USE and WITH clauses, RENAME statement, and the predefined environment). The entry catagories which are found within the symbol tables are:

1. Block, Subprogram, Package and Task Declarators
2. Symbol Entry and Fields
3. Type Entry and Fields

D.1.2.1 Block, Subprogram, Package and Task Declarators

The declaration of a block, subprogram, package or task generates a block of information which is attached to the visibility stack so that the proper context of the unit is represented (cf. Figure D-5). Enough data is kept in the declarator block to get to the symbols when they are in scope. For subprograms, information is kept that references the parameters; this information is used when processing a call to the subprogram.

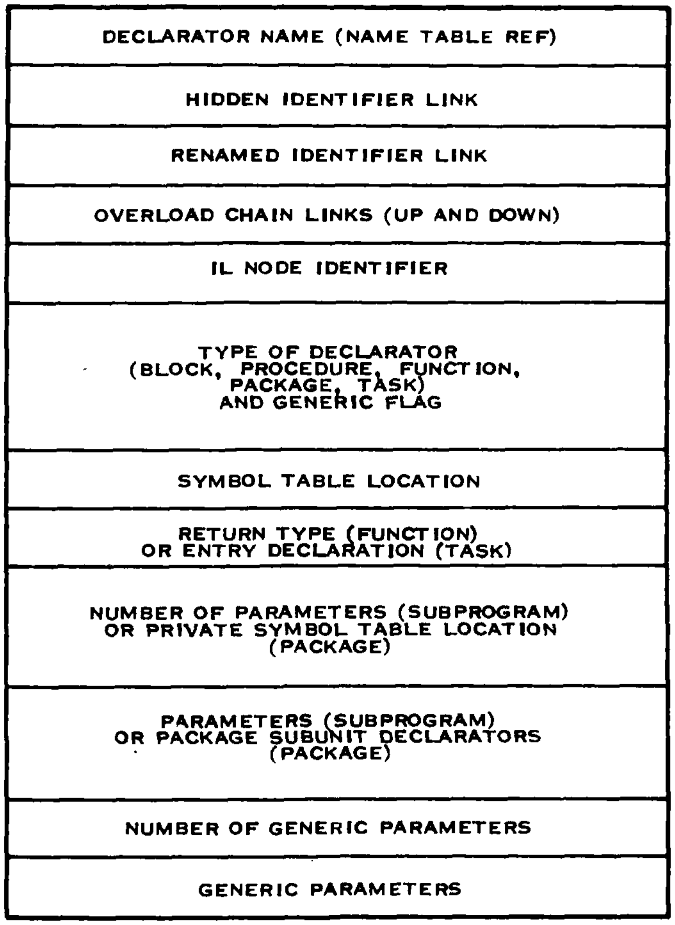

DECLARATOR NAME (NAME TABLE REF)
HIDDEN IDENTIFIER LINK
RENAMED IDENTIFIER LINK
OVERLOAD CHAIN LINKS (UP AND DOWN)
IL NODE IDENTIFIER
TYPE OF DECLARATOR (BLOCK, PROCEDURE, FUNCTION, PACKAGE, TASK) AND GENERIC FLAG
SYMBOL TABLE LOCATION
RETURN TYPE (FUNCTION) OR ENTRY DECLARATION (TASK)
NUMBER OF PARAMETERS (SUBPROGRAM) OR PRIVATE SYMBOL TABLE LOCATION (PACKAGE)
PARAMETERS (SUBPROGRAM) OR PACKAGE SUBUNIT DECLARATORS (PACKAGE)
NUMBER OF GENERIC PARAMETERS
GENERIC PARAMETERS

Figure D-5 Block, Subprogram, Package and Task Declarator

Declarator fields are:

1. Declarator Name -- Reference to the Name Table entry for the unit.

2. Type of Declarator -- This field indicates the type of the unit, i.e., block, procedure, function,package or task.

3. Generic Flag -- Set, if the unit is generic.

4. Symbol Table Location -- Virtual address of corresponding symbol table.

5. Return Type -- Reference to the result type of a function.

6. Entry Declaration -- Pointer to block of memory with copy of declaration of entry to task.

7. Number of Parameters -- Two part field to indicate number of parameters: how many are expected and how many are required.

8. Private Symbol Table Location -- Virtual address of private symbol table for packages.

9. Parameters -- List of parameter declarations

10. Package Subunit Declarators -- List of subunits in a package which are visible when the package is in scope.

11. Number of Generic Parameters -- Number of parameters which are expected and required to fill out the generic declaration at instantiation.

12. Generic Parameters -- List of generic parameter declarations.

The declarator block contains other information which is required for the successful elaboration of textual references, as found in selected component notation and overloading. These fields are described in Section D.1.2.2.

D.1.2.2 Symbol Table Entries and Fields

The Symbol Table Entries contain the working information of the symbol table. The entries are logically connected and the system of entries are eventually tied to the subprogram declarator of which they are a member.

Symbol table entries contain fields dictated by information required by the IL and internal information required to successfully determine proper operations (i.e., overloading and selected component elaboration) in the symbol table.

MY NAME (NAME TABLE REFERENCE)
I HIDE (HIDDEN IDENTIFIER THREAD LINK)
I RENAME (RENAMED OBJECT THREAD LINK)
I OVERLOAD (OVERLOADING CHAIN LINK)
OVERLOADED BY (RESERVE LINK ON OVERLOADING CHAIN)
MY IL (RESERVED VALUE OF IL NODE IDENTIFIER)
OBJECT CLASS (IDENTIFIES WHAT THIS NODE NAMES)
REQUIRED IL ATTRIBUTES (AS DEFINED BY SELECTED IL)

Figure D-6 Symbol Table Entry

Internal symbol table fields used are:

1. My_Name -- Reference to Name Table entry for the object.

2. I_Hide -- Thread to identifiers using the same name. This thread is used to:

   a. Establish uniqueness (when NIL).

b. Determining correct context for selected components.

c. Resetting Name Table reference when top entry is removed.

3. I Rename -- Thread to the symbol table entry of an object which is renamed by this entry. A non-nil pointer in this field also indicates that there is no IL data in the entry. This data is kept in the original entry. (cf. Section D.5)

4. I Overload -- Thread of overloadable identifiers using the same name. This thread is used to identify an outer scope entity which shares in overloading.

5. Overloaded By -- Reverse link of I_Overload.

6. My IL -- Reference to IL node generated for this Symbol Table entry. The generation of IL, for instance, for an inner entity may force the assigning of node identifications for global entities prior to the generation of the actual IL node for that global entity, in this case, a symbol table entry. This field may be used to reserve that node identifier for later IL generation.

7. Object Class -- This field identifies the nature of the object being represented by the symbol. The field may indicate:

a. Standard Object

b. Loop Parameter

c. Type Identifier

D.1.2.3 Type Table Entries and Fields

The Type Table Entries contain the working information of the type table. The entries are logically connected and the system of entries are eventually tied to the subprogram's symbol table and declarator of which they are a member.

Type table entries contain fields dictated by information required by the IL and internal information required to successfully determine proper operations requiring type checking (cf. Figure D-7).

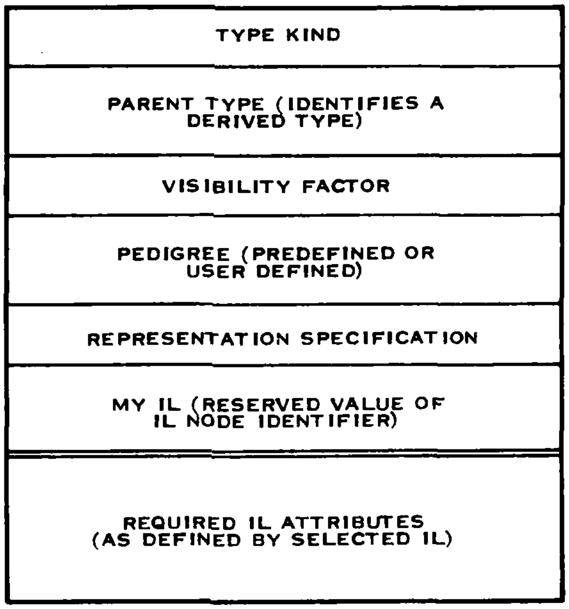

TYPE KIND
PARENT TYPE (IDENTIFIES A DERIVED TYPE)
VISIBILITY FACTOR
PEDIGREE (PREDEFINED OR USER DEFINED)
REPRESENTATION SPECIFICATION
MY IL (RESERVED VALUE OF IL NODE IDENTIFIER)
REQUIRED IL ATTRIBUTES (AS DEFINED BY SELECTED IL)

Figure D-7 Type Table Entry

Internal type table fields used are:

1. Type Kind -- This field identifies the nature of the type being represented by this entry. The field indicates the basic structure of the type, that is:

   a. Scalar Types : (Enumeration, Integer, Boolean, Character, Floating Point and Fixed Point)

   b. Array Type and Strings

   c. Record Type

   d. Access Type

2. Parent Type -- For use with the Derived Type Declaration. (cf. Section D.6).

3. Visibility Factor -- This field indicates the visibility of a type, used specifically with Private Types within a package which may be referenced in separate compilations.

4. Pedigree -- Indicates whether a type is predefined or user defined.

5. Representation Specification -- This is a reference to an IL node holding any representation specification declared for this type.

6. My IL -- Reference to IL node generated for this type table entry. The generation of IL, for instance, for an inner entity may force the assigning of node identifications for global entities prior to the generation of the actual IL node for that global entity, in this case, a type table entry. This field may be used to reserve that node identifier for later IL generation.

D.1.2.4 Virtual Symbol Table

It is very probable that all the symbol tables constituting the compilation context of a compilation may not fit into main memory. If the target machine has virtual memory, this is handled automatically. For those target machines without virtual memory, a software package shall be provided that implements virtual symbol tables. Virtualization of the symbol table portion of the symbol dictionary is easily attainable due to its disjunctive structure (which is a departure from the classic block mechansims due in part to the proper implementation of package declarations). When the symbol tables are loaded to create the compilaton context, symbol table references are assigned sequential virtual addresses. These virtual addresses are translated via page tables to the memory locations containing the entry. If the symbol table entry is not in memory, a page (block of symbol table entries), stored in a random file record, is brought into memory. If all pages are filled, a page is selected and written out to a random file record. A package shall be provided for accessing the symbol table entries. It keeps the actual mechanisms of reading and writing of records into paging areas transparent to the rest of the compiler.

D.1.3 The Look-Up Table

A Look-Up Table is kept for predefined and user defined identifiers (cf. Figure D-8). The Look-Up Table contains a reference to the Name Table entry containing the name of the identifier. Also maintained in the Look-Up Table is a reference to the DIANA NAME node corresesoponding to the name.

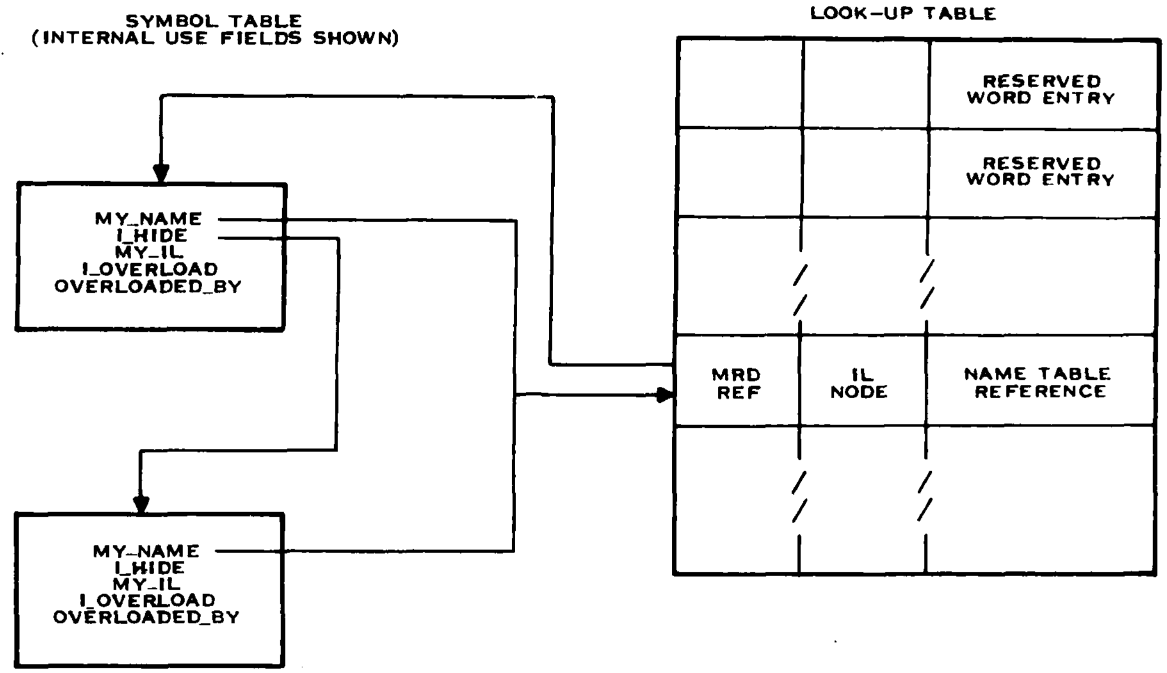


Figure D-8 Look-Up Table and Symbol Table Interface

The Look-up Table also contains a pointer to other entries within itself (not shown in Figure D-8) linking members of hash chains. When an identifier is placed into the Look-up Table for the first time, the character string which represents the identifier is hashed to get a numeric value. Identifiers which generate equivalent numeric values are chained together. This chaining restricts the set of identifiers to be searched in the Look-up Table in later operations.

The Look-Up Table for user defined identifiers maintains a 'Most Recently Declared' reference into the symbol table for each active name indicating the entry which last declared the name represented in the table. This reference is the anchor to the hidden names thread for all visible identifiers using the particular name. This thread is useful in finding overloading chains, resolving selected component references components and determining visibility of entities in a package referenced by a USE clause. However, most importantly, the 'Most Recently Declared' reference determines 'direct visibility'.

A sub-field to the 'Most Recently Declared' reference is a package declaration count which controls the direct visibility of some package identifiers in USEDed packages. The use of this sub-field is described in the Section D.3.2.

The initialization of the Look-Up Table takes place when the package STANDARD is elaborated and the 'Most Recently Declared' reference is set to this entry. Therefore, the names in the STANDARD package become directly visible at the start of the compilation of a unit until they are hidden and/or overloaded.

A separate portion of the look-up table is kept for predefined language attributes, predefined language pragmas, and reserved words. A call to a special look up subprogram to this portion of the table is triggered by:

1. A prime (') character in the proper context to denote an attribute

2. The reserved word PRAGMA

3. An expected reserved word as defined by the syntax.

The 'Most Recently Declared' reference for a reserved word or predefined identifier is a pointer to a symbol table entry containing its attributes. The symbol table for reserved words and predefined identifiers is not attached to the Visibility Stack.

D.1.4 The Name Table

The wide range of string lengths for identifiers is more easily maintained outside of the Look-Up Table in a Name Table. The Name Table contains the correct spelling of reserved words and predefined identifiers, user identifier names, and user defined numeric and character literals. Access to the Name Table is through the Look-Up Table name reference field.

D.2 The Predefined Environment

D.2.1 The Package STANDARD

All predefined identifiers, for example, those of built in types such as INTLGER, BOOLEAN and CHARACTER, operators and the predefined function ABS, are loaded into symbol tables as pre-compiled units of the package STANDARD [DoD80B, Appendix C] (cf. Figure D-9). The inclusion of the package STANDARD in the Visibility Stack serves as a base for the building of the compilation context. The location of the STANDARD package within the Visibility stack is covered in Section D.1.1.

Initialization of the symbol dictionary with the package STANDARD also consists of building entries within the Look-Up Table and Names Table. All identifiers declared in the visible part of the package STANDARD are assumed to be declared at the outermost level of every program. The connection of the 'Most Recently Declared' link from the Look-Up Table makes each of the visible identifiers in the STANDARD package directly visible. This visibility is equivalent to an implicit 'With STANDARD; Use STANDARD' prior to the translation of any user source code. In addition, the separately compiled subprograms and packages named in a WITH clause are assumed to be implicitly declared in STANDARD.

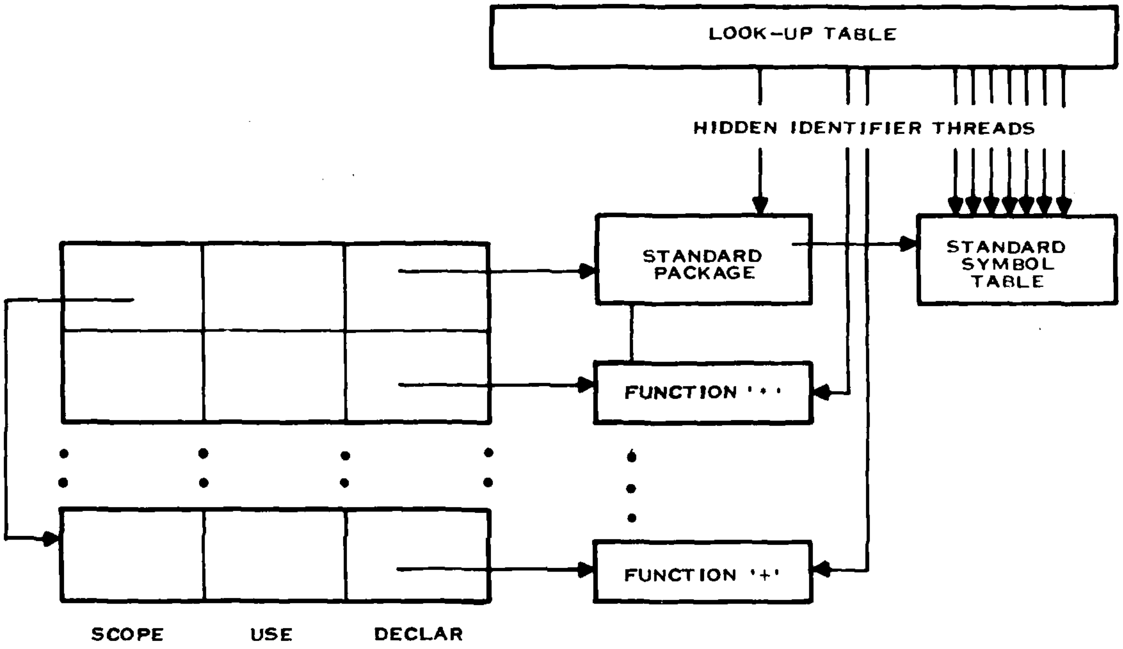


Figure D-9 Initialization of the Symbol Dictionary with STANDARD

D.2.2 Reserved Words and Pragmas

The Name Table and Look-Up Table are initialized to the reserved words and pragmas, and their symbol table loaded in order to present a consistent method of identifier recognition. Token numbers and hash chains related to an identifier's position in the Look-up Table are pre-assigned with this initiaization. The balance of information required for reserved words and pragmas is found in symbol table entries separate from other symbol tables and the Visibility Stack since their presence does not determine scope or visibility. These symbol tabl϶ entries may be referenced with the 'Most Recently Declared' pointer in the Look-up Table.

D.3 Maintaining the Compilation Context

D.3.1 Basic Block Structure Operations

The following sequences of operations provide the visibility of objects and proper compilation context necessary to successfully implement the Ada Rules of Visibility.

Entering Scope:

1. Get Subprogram Declarator Block and 'Push' its reference onto the visibility stack.

2. Update the 'Scope' entry in the visibility stack for the immediate parent of the subprogram to include the new declarator.

3. Fill in description of the subprogram into the declarator from the Ada declaration specification.

4. Attach the declarations of the parameters.

5. Get a block of empty symbols and types table entries.

6. Process symbol declarations of the subprogram.

   a. Hidden identifier thread extention

   b. Overloading chain insertions

   c. Add more symbol table blocks, as required

   d. Process and link 'Use' referenced packages.

Leaving Scope (Subprogram) :

1 Output IL for symbols and type table.

2. Remove symbols from hidden identifier thread.

3. Remove overloaded symbols from chains.

4. Dispose (or save in Libraries) blocks of symbols

5. Traverse 'Use' chain to disconnect packages, as required (More hidden threads and overloading).

6. Dispose of subprogram and package declarators on visibility stack between entry and 'Scope' reference.

7. The current subprogram stays with visibility stack until parent's scope is left.

Leaving Scope (Package) :

1. Output IL for symbols and types table.

2. Remove symbols from hidden identifier thread.

3. Remove overloaded symbols from chains.

4. Dispose (or save in Libraries) blocks of symbols

5. Traverse 'Use' chain to disconnect packages, as required (More hidden threads and overloading).

6. Remove references of subprogram and package declarators on visibility stack between entry and 'Scope' reference. The 'Scope' referenced region remains as part of the package as a reserved place for future USE clauses. The subprogram and package declarators within the package are chained to the package declarator for later access.

7. The current package declarator stays in the Visibility Stack until the parent's scope is left.

When the body of a package or subprogram is compiled, the symbol table for its corresponding specification is loaded (if not present).

The following sequence of diagrams and explanations illustrate the mechanics of manipulating the Visibility Stack and the symbol tables for the program in Figure D-10. Each of the diagrams show the 'classic' block structure symbol table and the Ada symbol dictionary in the same phase of development. This parallel development demonstrates that the Ada symbol dictionary operates in the same situations as the 'classic' table. Flexibility gained with this table structure is more apparent when coupled with the discussion on processing packages (cf. Section D.3.2).

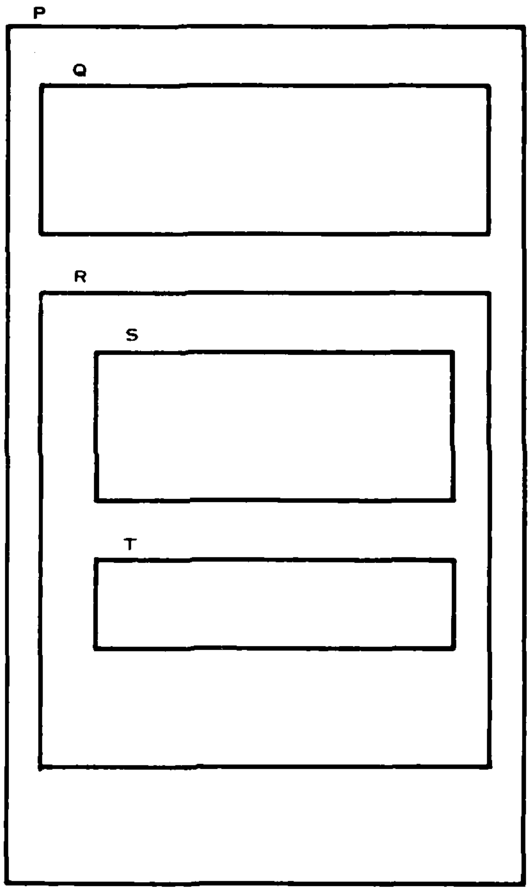


Figure D-10 Typical Block Structure

The subprogram P is entered and a subprogram declarator pushed onto the visibility stack. The end result of the processing of the declarations is shown in Figure D-11. The symbol table for P is attached to the declarator. Not shown, but required to provide proper visibility, are the connections to the Name Table in the form of hidden identifier threads, denoting direct visibility for the objects at the top of the stack.

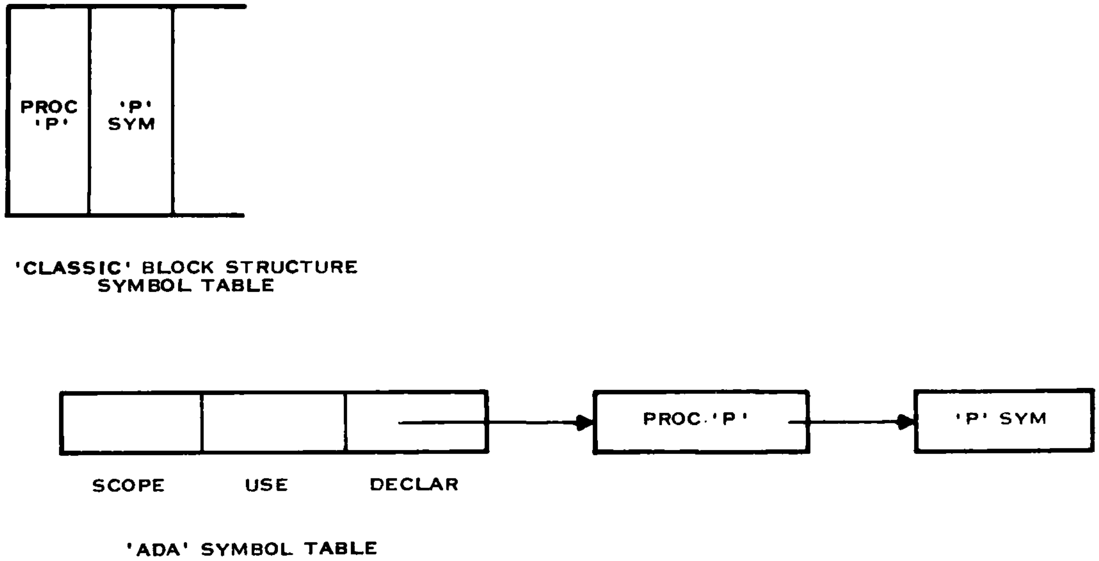
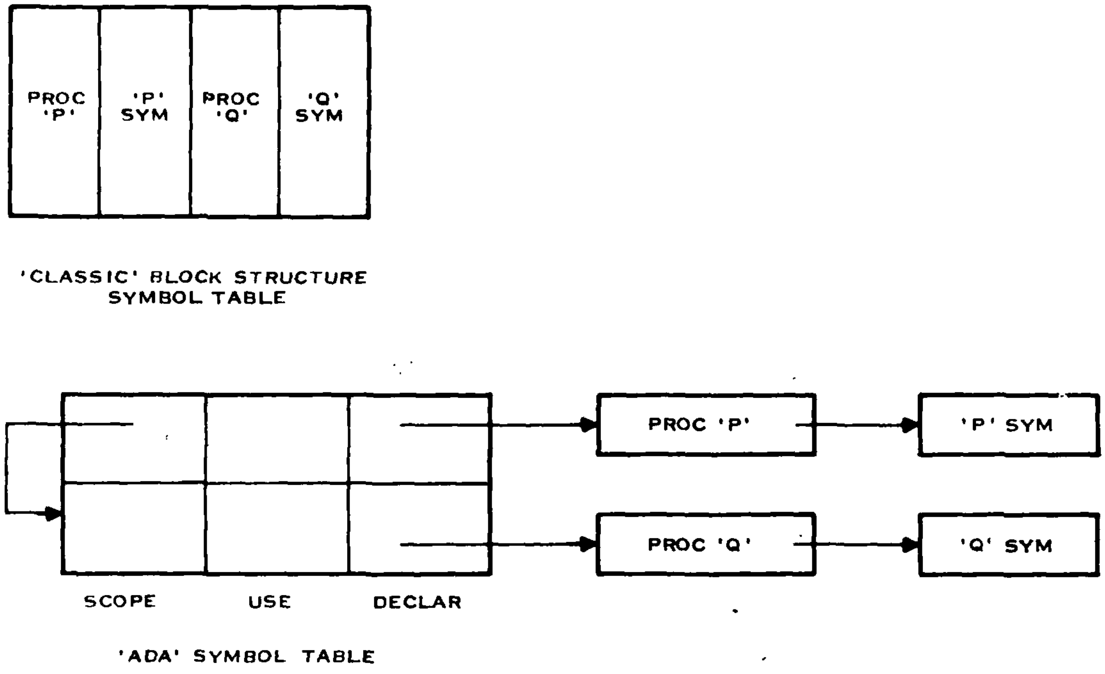


Figure D-11 Enter 'P'

The addition of the subprogram Q to the stack is illustrated in Figure D-12. The 'Scope' of subprogram P is extended to encompass the new subprogram. Once again, hidden identifier threads are established and any overloading declarations inserted into the proper chains.



'CLASSIC' BLOCK STRUCTURE SYMBOL TABLE

'ADA' SYMBOL TABLE

Figure D-12 Enter 'Q'

In Figure D-13, the scope of subprogram Q has been closed with exception of its callability. Its symbol table has been released and saved with the generated IL (cf. Appendix E, Section E.4.3).

After the closing of Q, the scope of subprogram R is opened. The declarator and symbol table of R are included within the scope of P. The 'Scope' entry for P is extended to show inclusion of R.

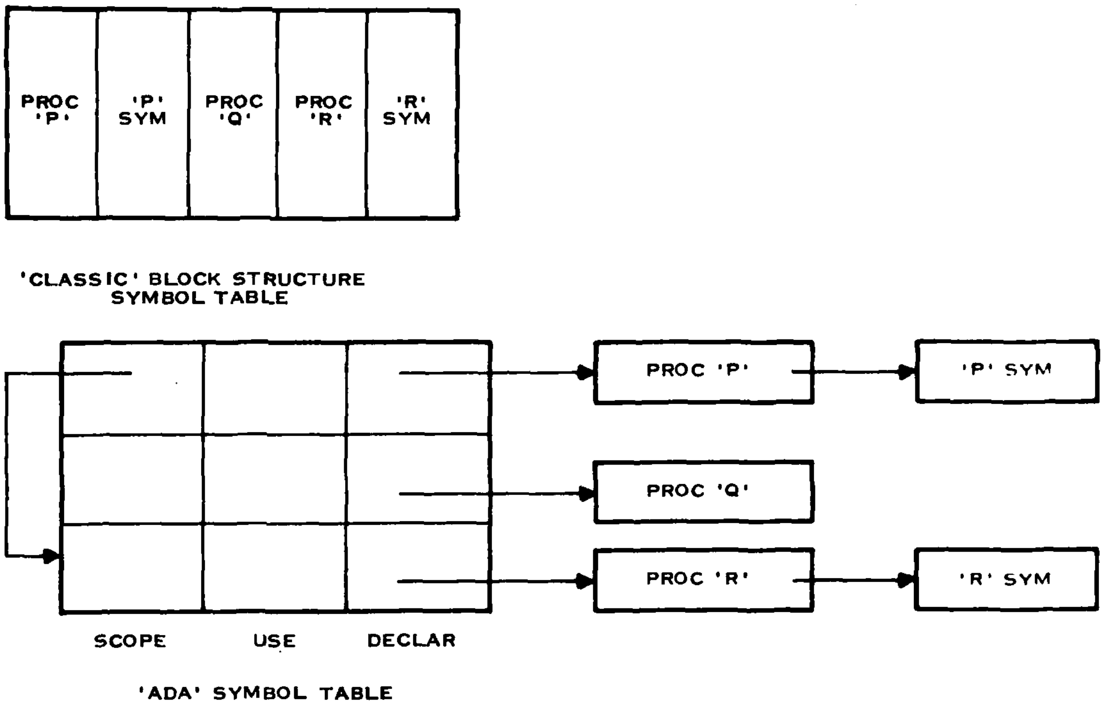


Figure D-13 Leave 'Q', Enter 'R'

Next the scope of R is expanded with the declaration of subprogram S (cf. Figure D-14). The 'Scope' entry for R is extended to include S.

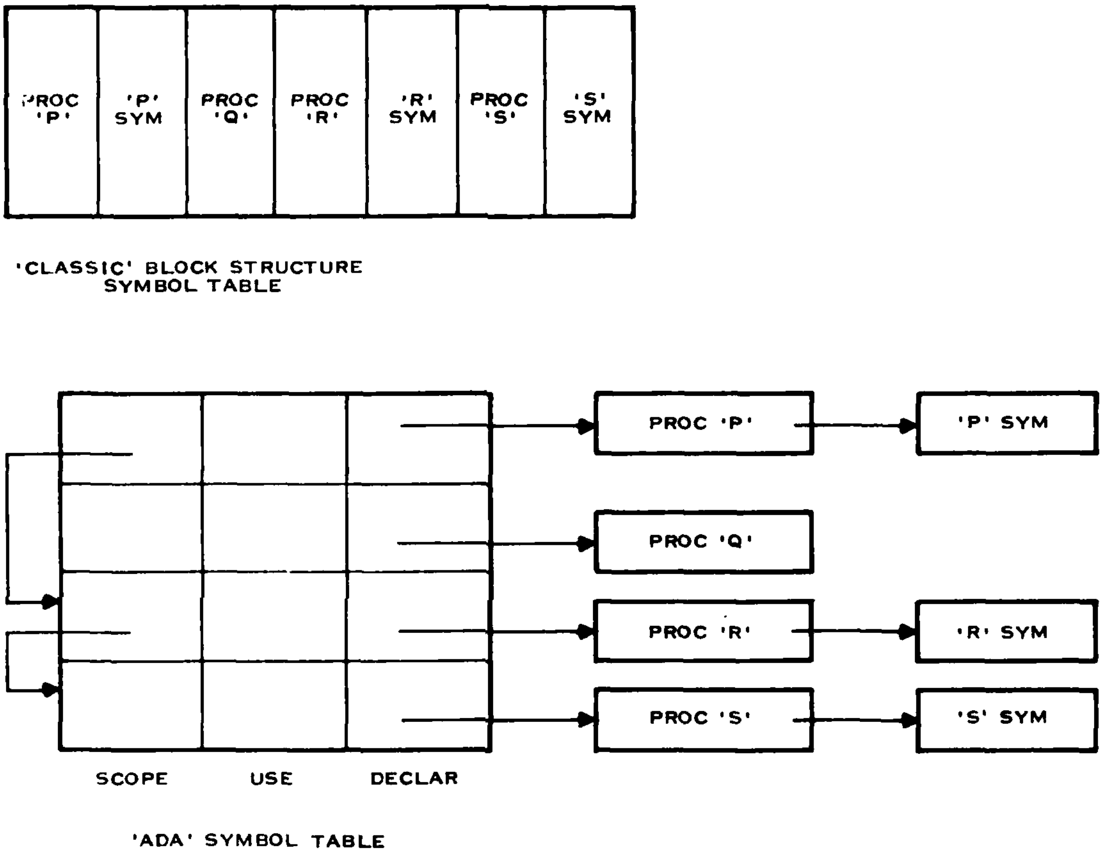
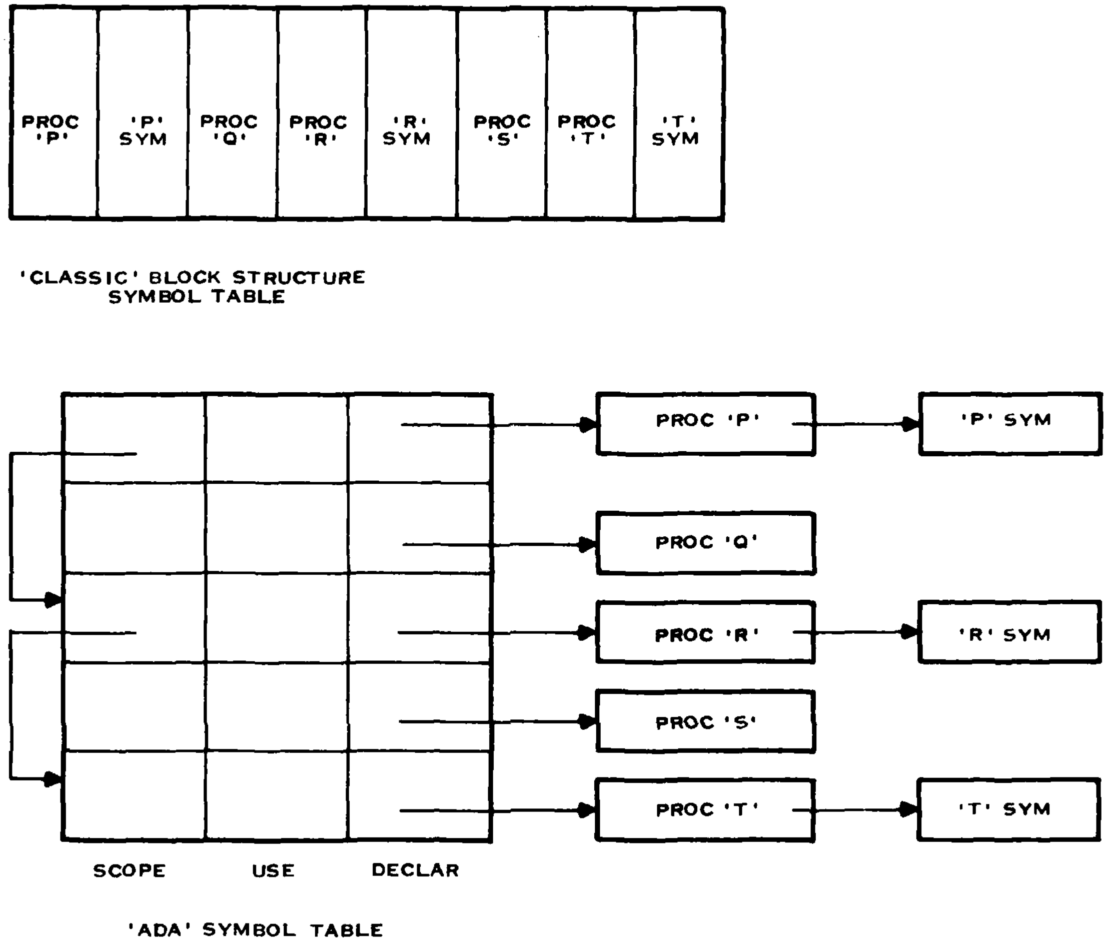


Figure D-14 Enter 'S'

Figure D-15 illustrates leaving the scope of S and entering the scope of T. The 'Scope' entry for R now encompasses the subprogram T. The subprogram declarator for S is still attached to the visibility stack for CALL information, but its symbol table has been de-allocated since its objects are no longer in scope.

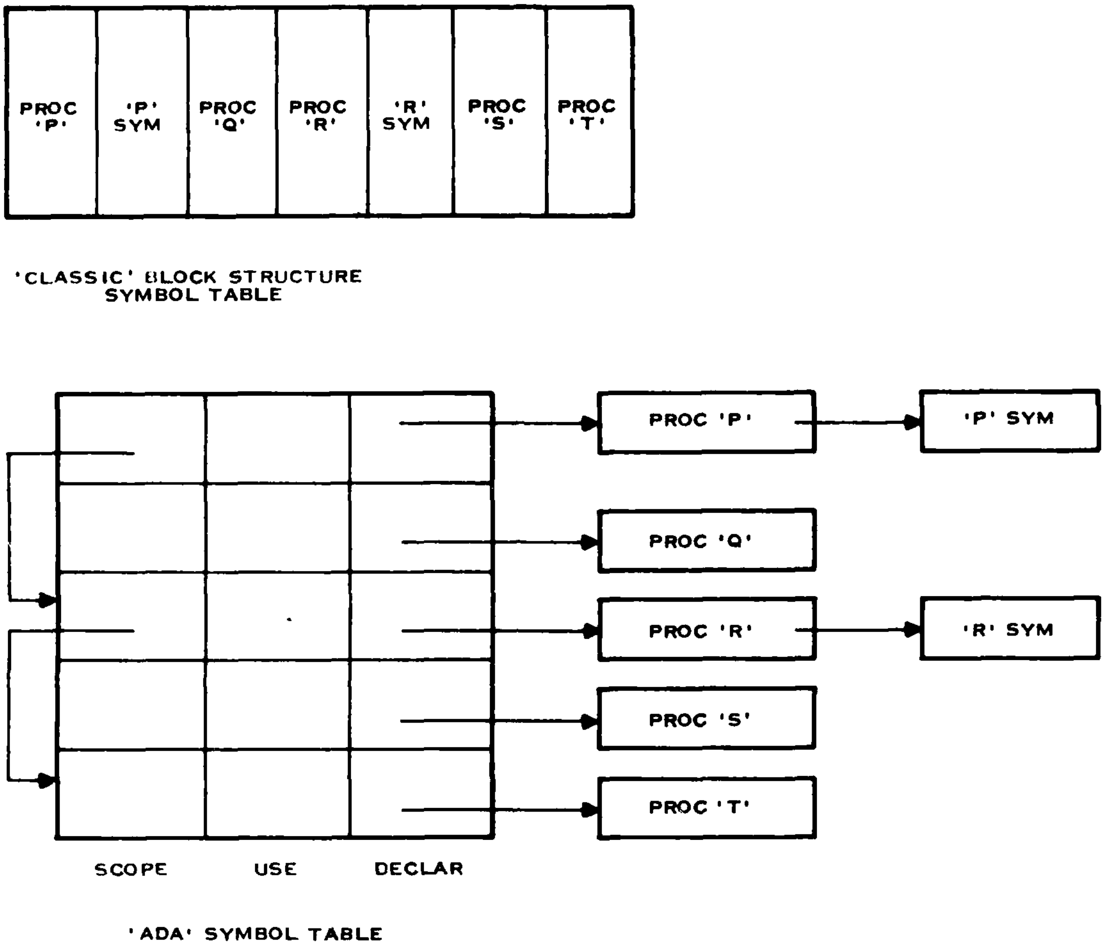


'CLASSIC' BLOCK STRUCTURE SYMBOL TABLE

'ADA' SYMBOL TABLE

Figure D-15 Leave 'S', Enter'T'

In Figure D-16, the subprograms S and T have been compiled within the scope of R and their declarators remain so that statements within R may call S and T. The body of R is then compiled.



'CLASSIC' BLOCK STRUCTURE SYMBOL TABLE

'ADA' SYMBOL TABLE

Figure D-16 Leave 'T', Still in 'R'

Leaving the scope of R is illustrated in Figure D-17. The steps involved in leaving its scope are:

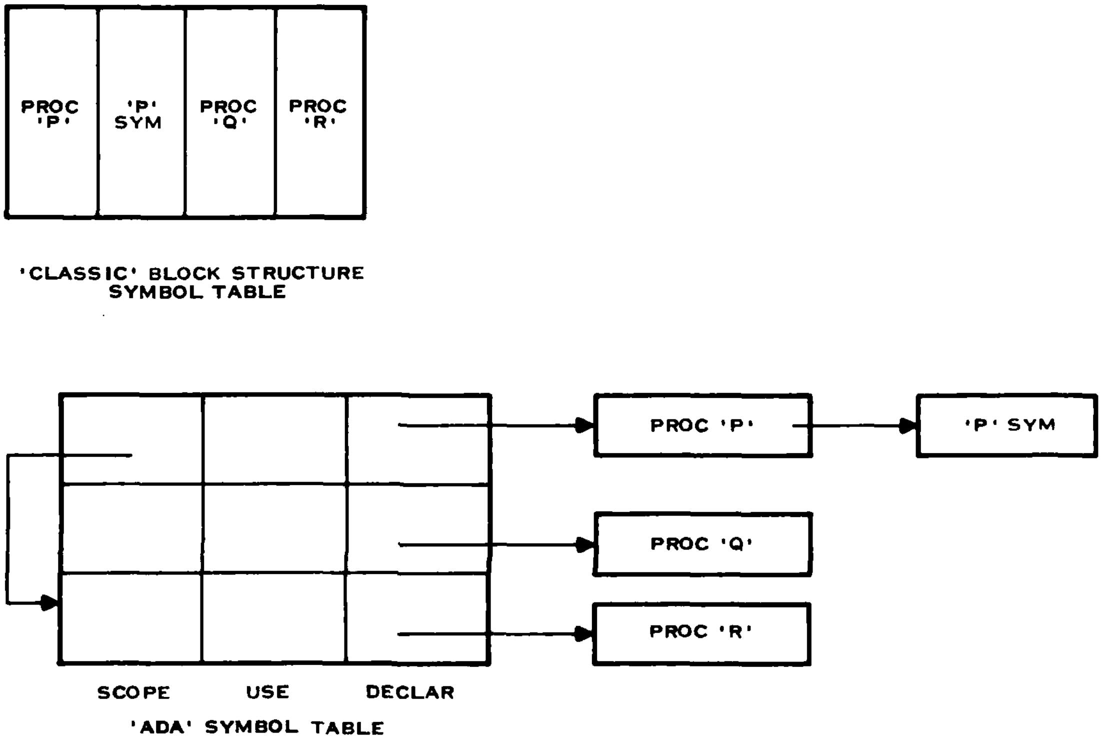
1. Save the IL and the symbol table for R.

2. Remove symbols from the hidden identifier thread. This operation consists of popping a linked-list stack.

3. Remove objects from the overloading chains.

4. Release symbol table blocks and disconnect reference from the subprogram declarator.

5. Deallocate the subprogram declarators bounded by the entry for R and the 'Scope' reference. This removes all mention of S and T.



'CLASSIC' BLOCK STRUCTURE SYMBOL TABLE

'ADA' SYMBOL TABLE

Figure D-17 Leave 'R', Still in 'P'

Leaving the scope of P brings the symbol table back to where it was when P was originally entered (cf. Figure D-18). Q and R are removed from the scope of P to bring the visibility stack to its original position. The symbol table of P is released, making its contents inaccessible from its brothers, but retaining its own callability.

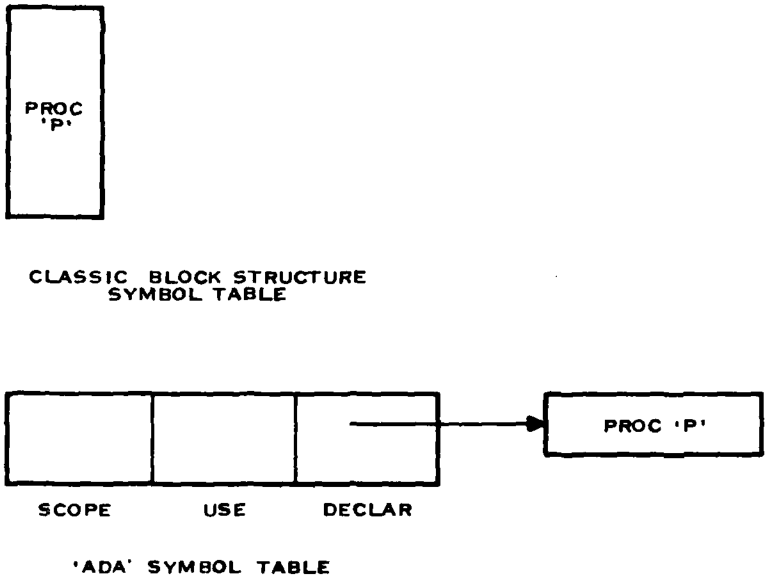


Figure D-18 Leave 'P', Still in Parent of 'P'

D.3.2 Processing Packages

The maintaining of the compilation context as detailed for a typical block structure requires an extention of those techniques for packages. Symbol table representations, scopes, and visiblities required to preserve the block environment and provide the proper methods to enforce the visibility of the package's parts, differ from the classical block structure referenced in the previous section. The presence of packages in Ada dictates the disjunctive nature of the symbol dictionary. It can be shown that these techniques in the presence of a package are totally compatible with the normal operation of the symbol dictionary in a block structure.

The following sequence of diagrams and explanations illustrates the mechanics of manipulating the Visibility Stack and the symbol tables for the package in Figure D-19.

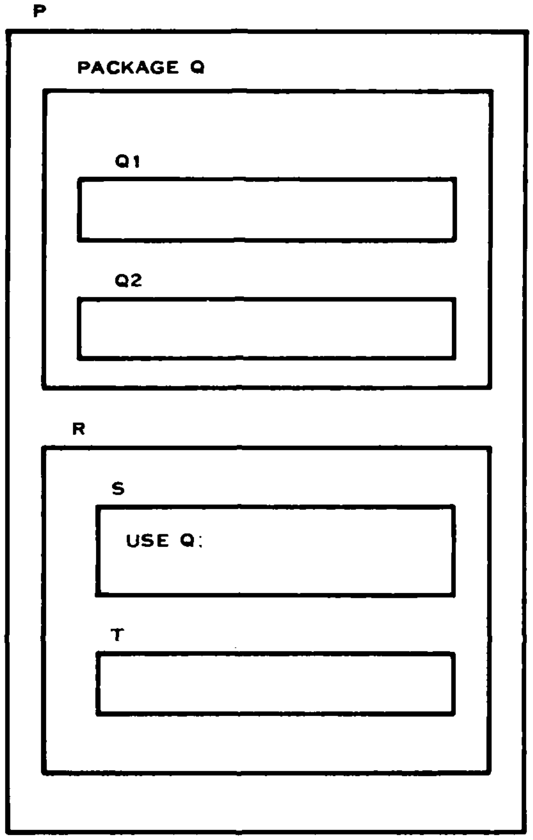


Figure D-19 Typical Block Structure With A Package

Packages allow the specification of groups of logically related entities. Generally, packages are used to describe groups of related objects and subprograms, whose inner contents are concealed from outer references. However, these private parts are fully available to members of the package during compilation. At compilation time, normal block structure operations are in effect for the package.

Consider the state of the symbol tables in Figure D-20. 'Q1' has been compiled and its symbols being out of scope, has had its symbol table detached. 'Q2' is in the process of being compiled. All objects, visible or private, of the package are directly visible at this time to 'Q2'. It is important to note the detachment of the 'private' portion of the symbol table from the visible portion. During the compilation of the package body, both symbol tables are loaded and are logically one. During the compilation of a unit that USEs the package, only the visible portion of the symbol table is loaded, thereby removing the private declarations from scope. When compiling a package body, the private portion of the symbol table is accessed through a pointer in the package declarator. Thus, direct visibility of entities in the private portion are provided according to the Ada visibility rules without benefit of special processing. This high level of visibility ends when the scope of 'Q2' and the package, 'Q', are left.

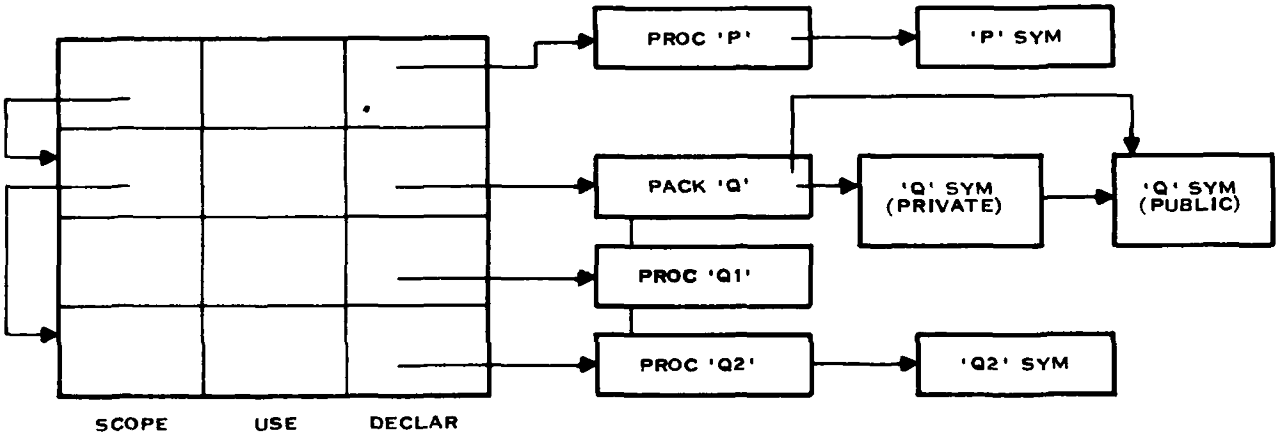


Figure D-20 In Package 'Q', Leaving 'Q1', Entering 'Q2'

The symbol dictionary allows the legal declaration of the same identifier twice in the same declarative part as long as the first declaration is a 'private' type in the visible part of the package specification and the second is a type in the private part. The 'Most Recently Declared' thread passes through both identifers so when scope is left and the private part of the package is removed, the visible declaration is still on the thread to keep it intact.

The central feature of processing packages is the inclusion of the package declarator in the Visibility Stack. This declarator serves as an anchor for package scope, selected component determination and reference point for future 'Use' chains. The package declarator also provides the central control of the package's private and visible objects, subprogram declarators and, in the case of generic packages, parameter declarations.

In Figure D-21, the scope of package 'Q' is left. This does not bring the total disconnection of symbol tables and removal of subprograms that the leaving of scope of a subprogram would entail. The concept of package demands that some remnant of the compilation context remain for use by the other elements of the program. The following items are accomplished to leave the compilation scope of a package:

1. Detach private symbol table from package declarator.

2. Ensure subprograms in a package are threaded to package declarator. This allows finding the subprograms through selected component notation.

3. For all subprograms within 'Scope' reference of package declarator, remove entries from the Visibility Stack. Retain the scope of the package declarator.

4. Remove symbols in visible symbol table from 'Most Recently Declared' threads in Look-Up Table. Names referenced in the Look-Up Table are not removed.

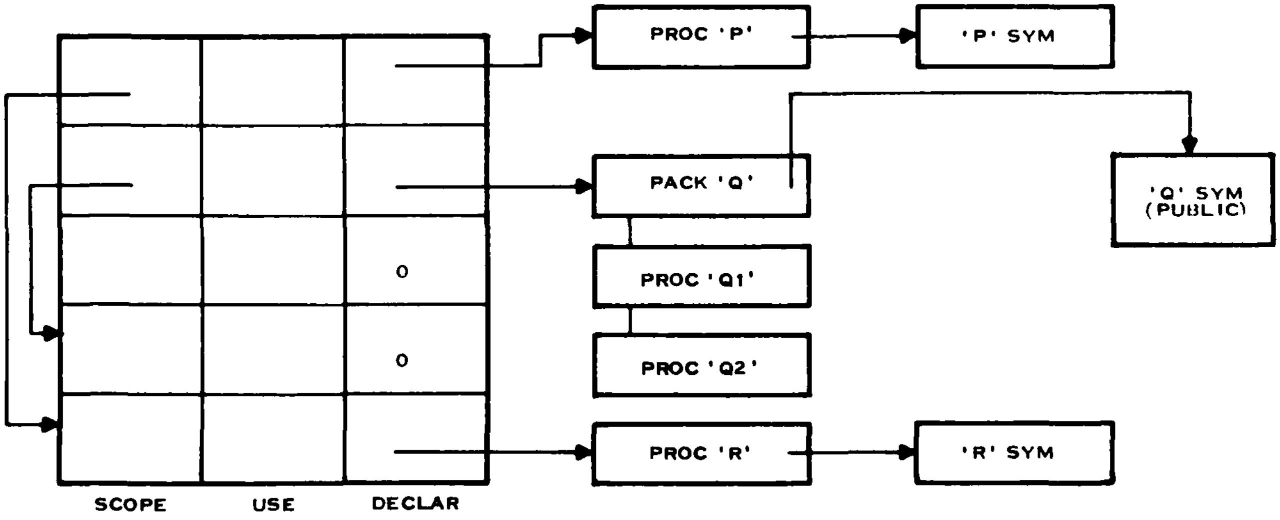


Figure D-21 In Proc 'P', Leaving 'Q', Entering 'R'

Figure D-22 illustrates the state of the symbol tables after the USE clause in procedure S is processed. The following actions are taken to provide direct visibility:

1. Objects in the package's symbol table are checked for eligibility as directly visible (cf. Section D.3.3). Eligible objects are connected with the Look-Up Table as 'Most Recently Declared'.

2. Subprograms visible within the package are also checked for direct visibility and overloading. Subprograms overloading other subprograms are included in the overloading chains. Directly visible subprograms are attached to the Visibility Stack in the entry reserved for them by the scope of the parent package.

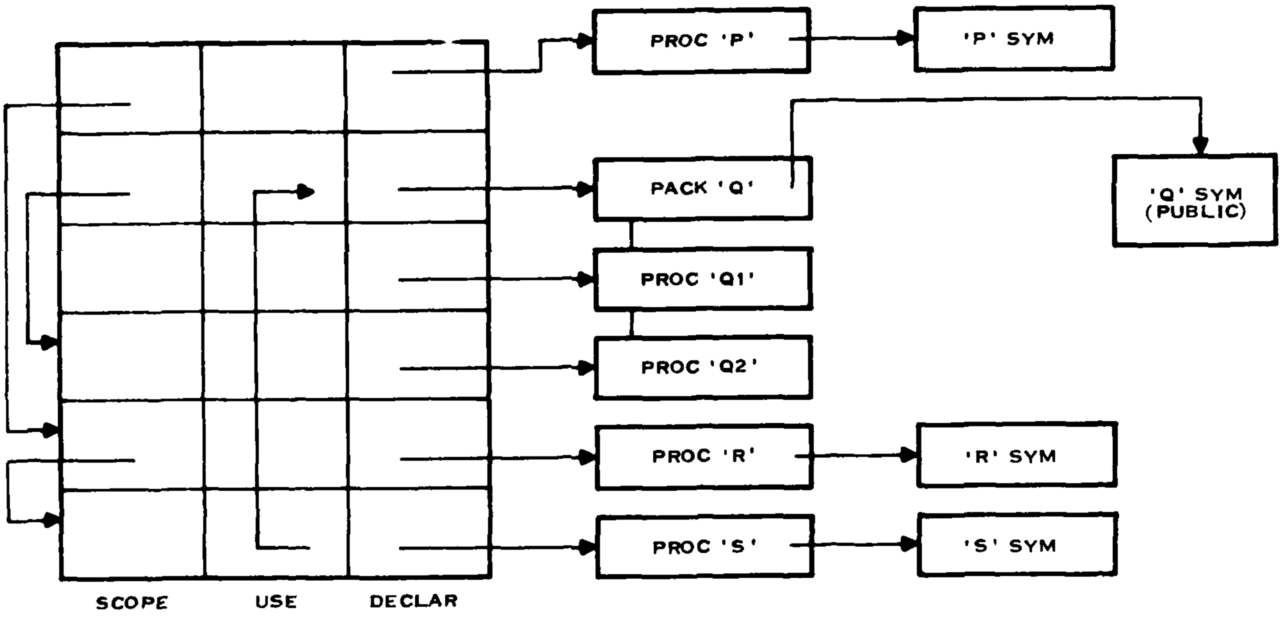


Figure D-22 In Proc 'S', 'Use Q'

D.3.3 Processing WITH and USE Clauses

The appearance of a library name in WITH clauses has the effect of including the specification for that library unit as an implicit part of the STANDARD package. Since the library name becomes part of the STANDARD declaration (already implicitly declared with a USE and WITH), the name becomes directly visible by being installed in the Look-Up Table and the 'Most Recently

Declared' thread attached. The thread which connects the children subprograms and packages of STANDARD is extended through the subprogram and package declarators of WITHed libraries.

For packages which are included with the STANDARD package through a WITH clause (cf. Figure D-23), their visible symbol table is included by placing a reference to it in the package's declarator. Also, entries in the Visibility Stack are reserved for each subprogram and package declared in the package's specification. When a USE clause denoting the package is encountered, the declarators for the visible subprograms and packages are linked to the Visibility Stack via their reserved entry.

The local objects of subprograms referenced by a WITH clause are not visible; only the symbol table for the subprogram's specification is loaded.

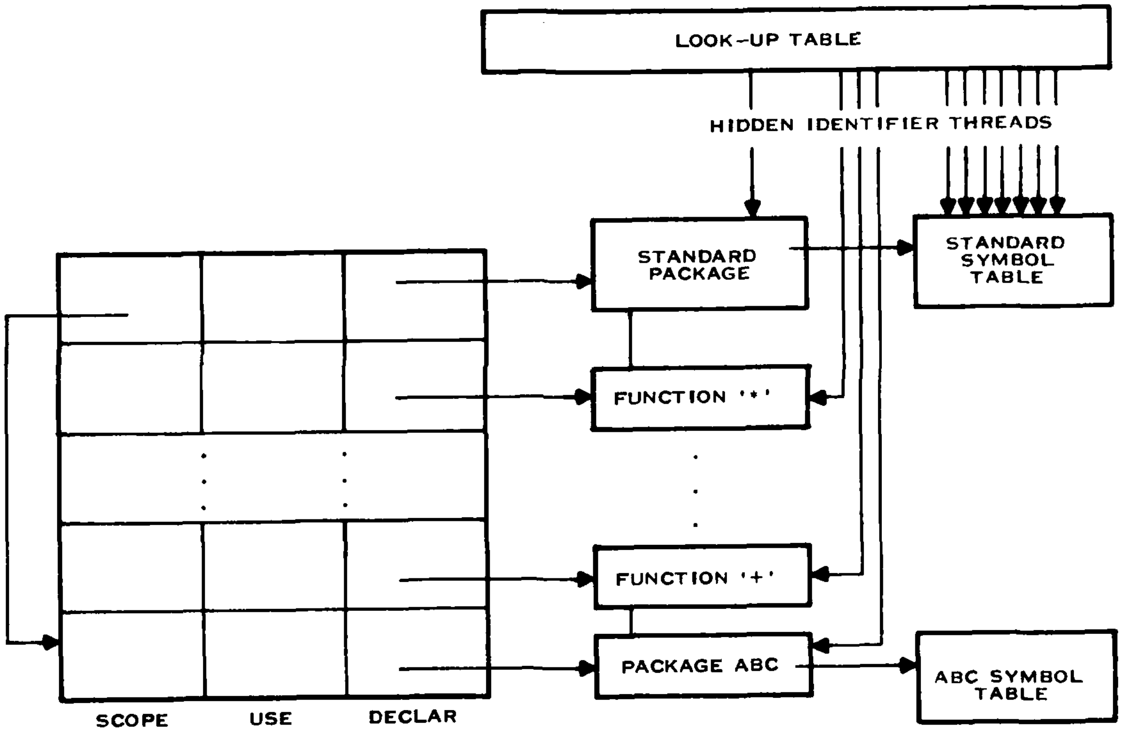


Figure D-23 WITH Package ABC

The inclusion of a package into STANDARD via a WITH clause does not provide direct visibility for the visible parts of the package, only the name of the package. A subsequent USE clause for the included package causes direct visibility of objects declared in the package's specificaton by setting the hidden identifier threads in the Look-Up Table accordingly (cf. Section D.4.5). The entry in the Visibility Stack for STANDARD is the base of the 'Use' chain to these packages (cf. Figure D-24).

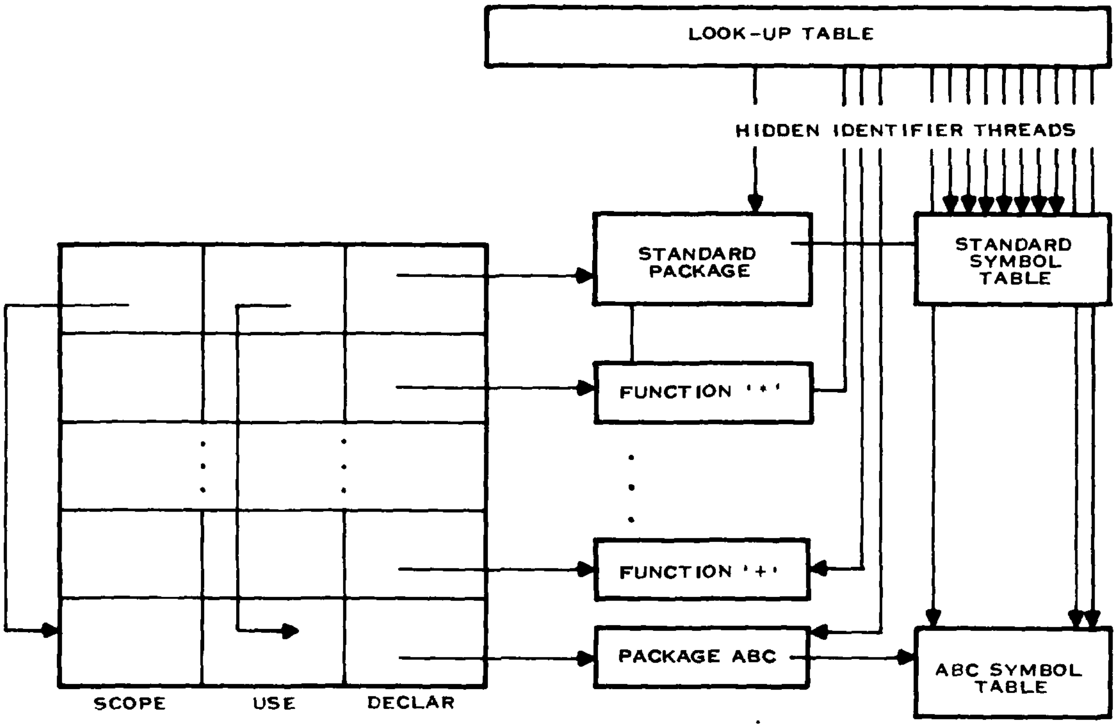


Figure D-24 USE Package ABC

D.4 Enforcing the Visibility Rules

D.4.1 Direct Visibility and Hidden Identifiers

The Ada Language Reference Manual [DoD80, Section 8.3] states, "The scope of the declaration of an identifier ... is the region of text over which the declaration has an effect. For each declaration, there exists a subset of this region where the declared entity can be named simply by its identifier; the entity, its declaration and its identifier are then said to be DIRECTLY VISIBLE from this subset." The mechanism of the 'Most Recently Declared' pointer keeps a constant reference to the symbol table entry representing the most inner declaration of an identifier. By finding the name of an identifier through the Look-up Table, the declaration directly visible is immediately available.

"An entity ... declared within a given construct is said to be HIDDEN within an inner construct when the inner construct contains another declaration with the same identifier. Within the inner construct the hidden outer entity is not directly visible." A further function of the Look-up Table's 'Most Recently Declared' pointer is to anchor the thread of hidden identifiers. The thread operates as a stack structure with the top entity being directly visible and other declarations of the same identifier in outer scopes being pushed deeper into the stack. When an inner scope is left, the top entity is popped from the stack allowing the next innermost identifier to become directly visible again.

D.4.2 Loop Parameters

The Ada Reference Manual [DoD80, Section 5.5] states, "The execution of a loop statement with a for iteration clause starts with the elaboration of this clause, which acts as the declaration of the loop parameter... the loop parameter is declared as a variable, local to the loop statement...Within the basic loop, the loop parameter acts as a constant." The occurence of a for iteration clause generates an anonymous block declaration in the symbol dictionary. The Visibility Stack references a block declarator with no name, no parameters and a single entry symbol table. The symbol table entry declares the loop parameter. This entry has the object class of loop parameter so when the parameter is used within the scope of the loop, it can be treated as a constant for computation, but properly incremented and tested as a variable at the end of the loop's sequence. The loop parameter is directly visible during the scope of the loop, hiding any other declaration of the same identifier in any outer scope.

D.4.3 Overloading and Operator Identification

The Overloading Chains (cf. Figure D-25) are mechanisms set up by the opening of the scope for a subprogram facilitating the resolution of otherwise ambiguous subprogram calls and enumerated literal usage. The overloading algorithms require a means of checking all members of a particular chain, so the chain is doubly linked.

Members of an individual chain belong to one of the two overloadable classes, subprogram and enumerated literals. The chains of the classes are not shared, even though the identifier is the same. However, the hidden identifier thread which links all identifiers of the same name makes it possible to get from one class chain to the other along that thread.

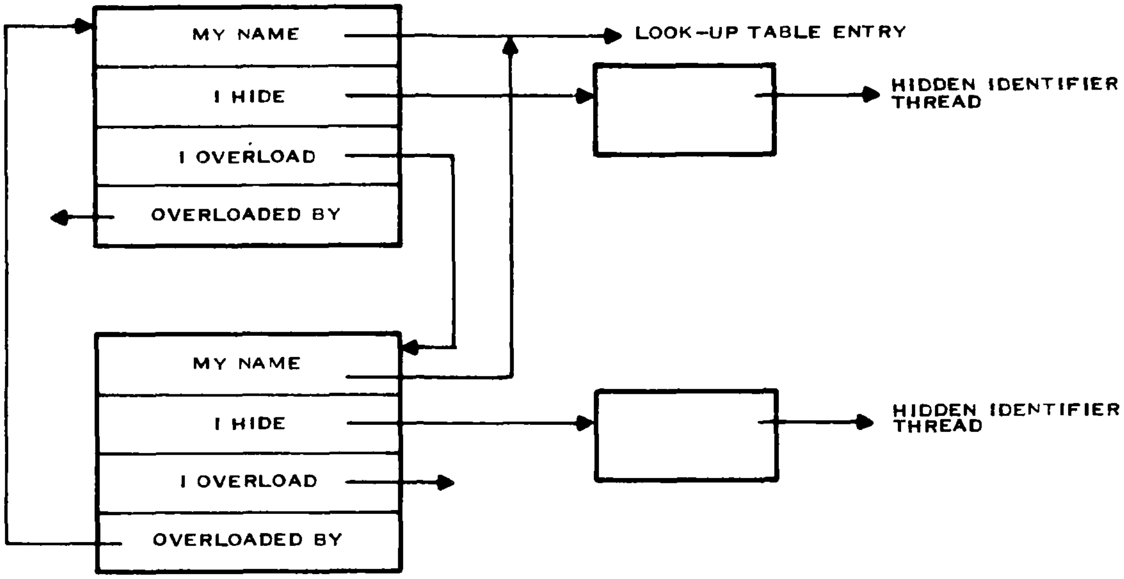


Figure D-25 Overloading Chains in the Symbol Table

It is the responsibility of the Operator Identification algorithm to determine, from the syntax and semantics, the appropriate operator, function or subprogram in an overloaded condition. The Ada Rationale [ICH79] states a multi-pass algorithm is required for this determination; contextual information is propagated in both a top-down and bottom-up traversal of the expression tree until convergence is achieved. However, it has been shown [PEN80, PER80] that two passes are sufficient to resolve the identification of the operation(s). The Pennello/Deremer/Meyers algorithm [PEN80] can be fully implemented with the symbol table structures, as designed. For each operator, a set of overloaded operators is constructed. Based on the operand types required by the parent operator and available operand types of children operators, or the parameters of subprograms (i.e., types, order, number), nonapplicable members of the set are eliminated. The result is a single member set for each operator indicating the correct operator to apply.

The implementation of the algorithm uses an array of BOOLEAN variables to indicate operator membership in the set (cf. Figure D-26). The tree node being visited by the algorithm references the head node in the chain of overloaded operators, functions or procedures. An available operator is recognized by its 'TRUE' value in the array. The operator is then located by its position in the boolean array, and its corresponding relative location in the overload chain.

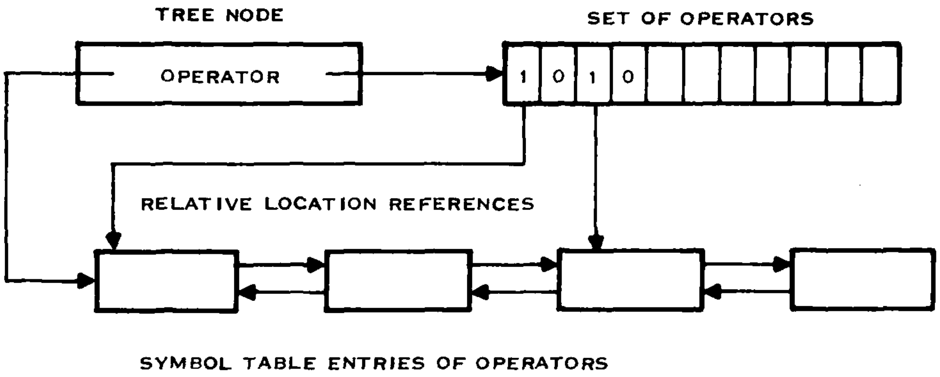

TREE NODE
SET OF OPERATORS
OPERATOR
1
0
1
0
RELATIVE LOCATION REFERENCES
SYMBOL TABLE ENTRIES OF OPERATORS

Figure D-26 Operator Identification Structures

D.4.4 Selected Components

The strategy of the symbol table provides a method of determining the identity of objects named as a selected component. The Ada Reference Manual [DoD80, Section 4.1.3] states, "Selected components are used to denote record components. They are also used for objects designated by access values. Finally, selected components are used to form names of declared entities... A selected component can denote either

a. A component of a record...

b. An object designated by an access value...

c. An entitiy declared in the visible part of a package...

d. An entry (or entry family) of a task...

e. An entity declared in an enclosing subprogram body, package body, task body, block or loop..."

The algorithm (cf. Figure D-27) locates an object by its selected component name. With the exception of the last name in the selected component, all identifiers in the selected component must have symbol table entries referencing other identifers or target types. As long as the names in the selected component agree with entries in the symbol tables and subsymbol tables (for instance, record field descriptors), the search is successful. A completed search returns an appropriate symbol table entry. An unsuccessful search resets the pointers to the selected component name list and re-start the search with the next instance of the first name along the hidden identifier thread.

```
Find first name in selected component in Look-Up Table
While there are more entries along hidden identifier thread
   Follow hidden identifier thread to next entry
   While there is success in locating names in symbol tables
      Case of type of entry is one of the following:
         Access -- Type points to simple type or structured type
         Record -- Type points to structured type
         Subprogram -- Declaration points to symbol table and/or
                           other subprograms, packages and tasks
                           within scope
         Package -- Declaration points to symbol tables and/or
                           other subprograms, packages and tasks
                           within scope
         Task --  Declaration points to declaration of entry point(s)
      Otherwise Exit while loop as unsuccessful
      Locate symbol table entry referenced by type
      Check symbol table entries for next component name
          (other subprograms, packages and tasks for
           subprogram and package declarations)
      If not found -- Exit loop
          Reset selected component name pointer
          Exit while Loop
      If last name in selected component
          Exit while loop as successful
   Endwhile
   If Successful name found--
      Exit while loop as successful
Endwhile
If selected component not found--Return nil symbol table pointer
```

Figure D-27 Selected Component Identification Algorithm

D.4.5 Direct Visibility for Package Entities via USE Clause

When a USE is elaborated, the visible symbol table of the package is scanned. An identifier having a name not appearing in the Look-Up Table is entered into the Look-Up Table as any other identifier. An occupied Look-Up Table 'Most Recently Declared' reference denies the direct visibility to the package's identifer [DoD80B], Section 8.4] (Use Clause Rule 1).

The mechanisms of the symbol tables and the 'Most Recently Declared' reference of the Look-Up Table must be able to handle the visibility rule concerning the repeated declaration of an identifier in a second used package. The Ada Reference Manual states that an identifier must be declared in the visible portion of ONE AND ONLY ONE package. Therefore, the direct visibility of a used package's variables must be removed, but should be restored with the leaving of the scope of the second package. Using the action table in Table D-1, the symbol tables, and the look-up table, this rule can be effected.

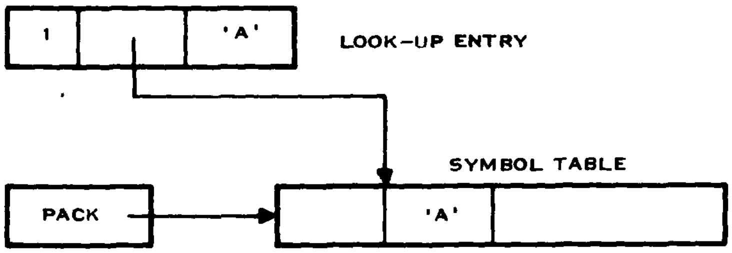
Table D-1 USE Visibility Determination Table--Entering Scope

0	NIL	ATTACH SYMBOL ENTRY TO MRD INCREMENT PACKAGE COUNTER
0	NOT NIL	IDENTIFIER IS IN USE DO NOT ATTACH PACKAGE ENTRY
>0	NIL	ERROR
>0	NOT NIL	IDENTIFIER IS USED BY ANOTHER PACKAGE, INCREMENT PACKAGE COUNTER DO NOT ATTACH PACKAGE ENTRY

When a USE clause is encountered, the visible identifiers of the package are located in the Look-Up Table. Consider the case when the Look-Up Table entry for an identifier, 'A', is:

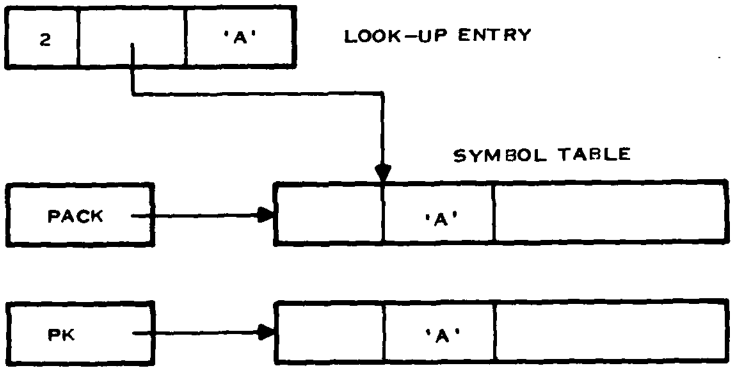
0	NIL	A

Applying the action found in Table D-1, the identifier is made directly visible by setting the 'Most Recently Declared' reference in the Look-UP table, and the package declaration count is incremented to show the presence of a USE clause. The completed action leaves the Symbol Table and Loop-Up Table as follows:



SINGLE PACKAGE DECLARATION OF AN IDENTIFIER

The second occurrence of the identifier, 'A', within another package designated by a USE clause invalidates the direct visibility of the first occurrence of the identifier. Using the action directed Table D-1, the package declaration counter is incremented to show the second usage of the identifier. The value of the counter now indicates that the MRD reference does not indicate direct visibility. The state of the tables is now:



MULTIPLE PACKAGE DECLARATION OF AN IDENTIFIER

At any time, it is possible for the identifier, 'A' to be declared in a subprogram's declarative part. Using the normal algorithm for processing the declaration, the new 'A' reference is now the 'Most Recently Declared' and directly visible. Its I_Hide entry contains the reference to the package 'A' and the package count. The package count in the Look-Up Table is cleared to zero. Note, this combination does not allow the attachment of any package identifiers to the name, 'A'. When the scope of the subprogram's 'A' is left, the original package designators are restored to the Look-Up Table.

Another set of actions are applied when the scope of a package is left. These are shown in Table D-2. These actions properly restore the compilation context and visibility of the outermost package.

Table D-2 USE Visibility Determination Table--Leaving Scope

PKG COUNT	MRD VALUE	ACTION
0	NIL	ERROR
0	NOT NIL	IDENTIFIER WAS IN USE, PACKAGE ENTRY WAS NOT ATTACHED, DO NOTHING
>0	NIL	ERROR
1	NOT NIL	IDENTIFIER USED BY THIS PACKAGE ONLY, DECREMENT PACKAGE COUNTER, DETACH ENTRY
>1	NOT NIL	IDENTIFIER IS USED BY ANOTHER PACKAGE, DECREMENT PACKAGE COUNTER, PACKAGE ENTRY WAS NOT ATTACHED

The techinque used for Rule 1 also applies to enumeration literals when the literal is an identifier. Character literals are to always be visible [DoD80B, Section 8.4] (USE Clause Rule 2). See Section D.4.3 for implementation of overloading of enumeration literals from a package.

A subprogram in a package may be made directly visible if it does not have the exact structure of a visible subprogram (traverse the hidden identifier thread checking other subprograms with subprograms in the package specification) and it may not be visible if any non-overloadable entity exists with the same name (stop traversal of hidden identifier thread if a non-overloadable entity is found) [DoD80B, Section 8.4] (USE Clause Rule 3). See Section D.4.3 for implementation of overloading of subprograms from a package.

D.5 The RENAME Statement

For processing the RENAME statement, the two design choices considered were:

1. Create a thread for the new name from the Look-Up Table through the renamed node. However, multiple renames of the original name results in countless threads through the symbol table entry which

is unmaintainable. Also, renaming of the new name is difficult to implement and maintain.

2. Create a new symbol table entry for the new name. However, there remains the determination of where the IL data for the new symbol table entry should be kept. Since the representation of the IL is subject to changes after a rename, i.e. representation specifications, etc., if the IL were copied into the new symbol table entry, there is the possiblity that changes to the IL may not be incorporated in the renamed entry. Therefore, it has been decided to create a new symbol table entry and link it to the original for IL information. This design also allows independent overloading and hiding of the renamed object.

The second choice was selected because of its ease of implementation and maintenance.

"Renaming may be used to resolve name conflicts, to achieve partial evaluation and to act as a shorthand." [DoD80B, Section 8.5]. The selected method of representing a renaming in the symbol table covers the resolution of name conflicts. However, some extra processing is involved in partial evaluation and shorthand. The following statements demonstrate the use of these concepts:

1. XYZ renames A(5);

2. XYZ renames M.N.O.P; (where all components are records and/or record fields -- no access types)

3. XYZ renames A(L);

4. XYZ renames M.N.O.P; (where N is an access type)

Statements 1 and 2 are considered 'static' renaming statements since the type of the variable and its relative location in the run-time stack can be determined at compile-time. On the other hand, statements 3 and 4 are considered 'dynamic' renaming statements because the exact location in the run-time stack structure must be evaluated at run-time from the current value of L in A(L) and the allocated storage of N in the selected component expression, respectively. The static and dynamic statements are handled the same in the symbol table, i.e., the entry in the symbol table for the renaming object references the renamed object (cf. Figure D-28), but differently during code generation.

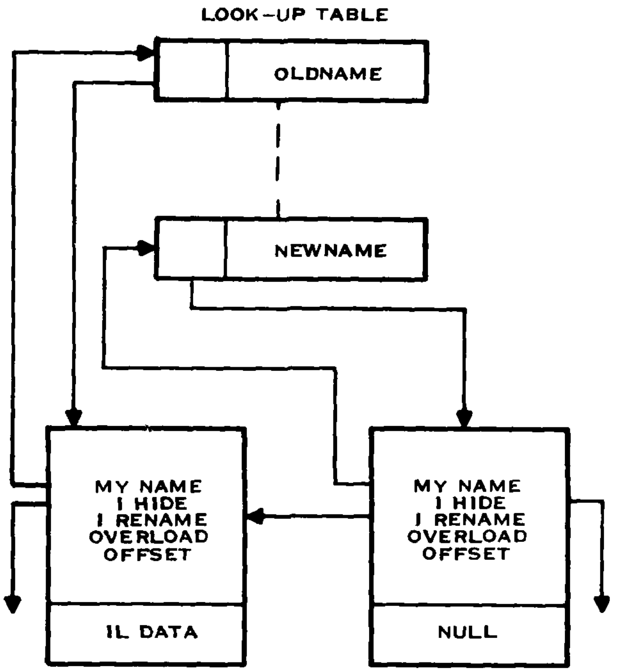


Figure D-28 Newname RENAMES Oldname

In static renaming statements, the reference to the renamed object exists primarily to indicate the stack frame offset (determined at compile-time) of the renamed object and typing informaton about the renamed object. Generated code accesses the renamed object directly.

In dynamic renaming, the run-time stack frame offset cannot be calculated at compile-time. In this case, the renaming object is allocated space in the stack frame for the subprogram in which it is declared. This entry is an indirect reference to the renamed object, so that source code use of the renaming object results in code using the local indirect reference. The contents of the renamed object is determined at entry into the subprogram. Although the renamed object making the statement dynamic may change after the elaboration of the RENAME, the indirect reference remains static as long as the RENAME is visible.

D.6 Derived Types

A derived type has its own symbol table entry which is a copy of the parent type with any declared constraints. A reference to the parent type through a Derived Base Type field serves as link in determining type suitability for derived subprogram parameters and return values.

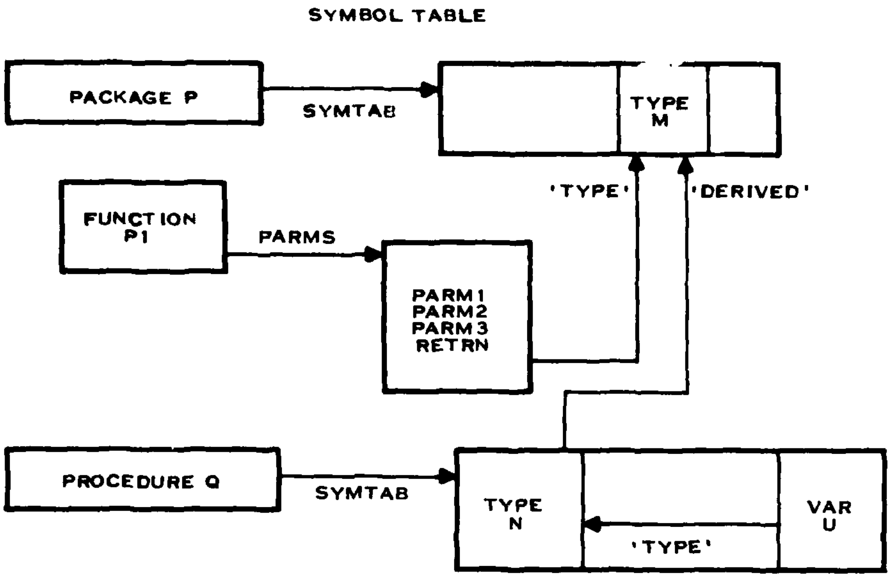


Figure D-29 Linkages for a Derived Type

Figure D-29 illustrates the state of the symbol table for the following code:

```
Package P is
  Type M is ........;
  Function P1 (....) Return M;
End Package P;
Subprogram Q is
  Use P;
  Type N is new M;
  U is N;
Begin
    :
  U := P1(...);
    :
End Q;
```

According to the rules governing derived types, the object 'U' is a legal return parameter in the function 'P1'. This legality can be found using the following algorithm:

```
      Find type of object used as parameter.
   2: Find type of expected subprogram parameter.
      If types disagree
         If checked type is a derived type, get derived
            base type and re-enter Algorithm at label 2
         else return a Failure For type check.
      If types agree, Return with Success for type check
```

This algorithm and derived type chaining allows the use of predefined functions for all types derived from STANDARD and predefined types corresponding to those functions.

APPENDIX E

REPRESENTATIONS OF DIANA

E.1 Design Issues

In order to process DIANA effectively, a number of design decisions must be made. Certain design issues involve defining the implementation-defined types used in the definition of DIANA, in particular, the source position of a node and the representation of identifiers. (The latter is described in Appendix D.) Other design issues involve the representations of DIANA. There are three basic representations of DIANA (cf. Figure E-1): the internal representation, the external representation, and the visible representaton.

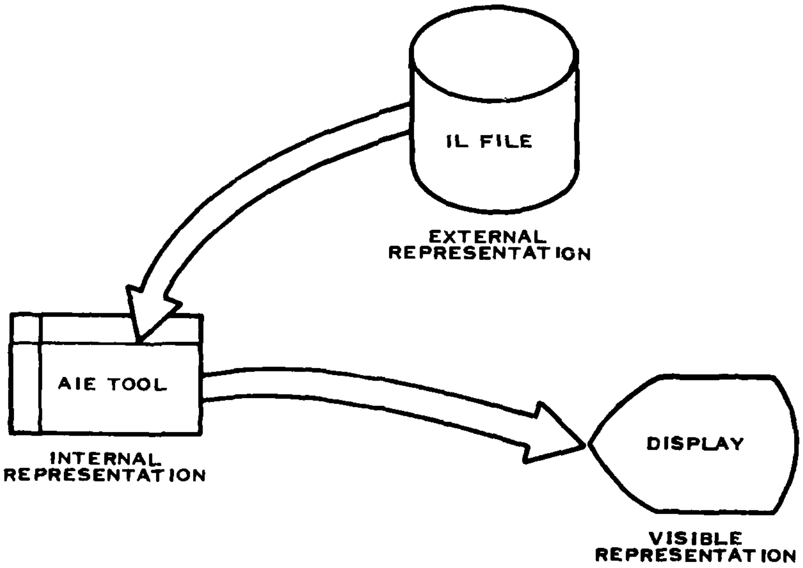


Figure E-1 Representations of DIANA

The internal representation is the representation of DIANA in main memory. The external representaton is the representation of DIANA on a file. The visible representation is the human readable representation of DIANA.

Finally, there are design issues concerning the transformation of the the external representation to the internal representation, and the derivation of new dialects.

Note, the design decisions described below on external representation, symbol tables, Source Table, etc. constitute a dialect of DIANA. The designers of DIANA intended for DIANA to be extensible to different dialects, i.e., for the DIANA definition to be specialized by making implementation decisions on the private types used in the definition, and for attributes to be added for tool specific information. The design decisions made are well within the framework of the dialect derivation process intended by the DIANA designers.

E.2 Source Position

There are cases where it is necessary to relate a node back to the source text from which it was created. For example, the source position is used during semantic analysis to report error messages. Also, it is used by the source level debugger to relate debug information back to the source text; this is particularly important if optimization has been performed. The position in the source text is uniquely specified by a line number and character number within the line. Because of the INCLUDE pragma it is also necessary to uniquely identify the source file. Therefore the value of the source file attribute is the triplet (source_file, line, char) which is collectively called the "source_designator".

The "source_file" is the database name of the source file which uniquely specifies the revision and version. The "line" part is an integer; source lines in a file are numbered sequentially starting at 1. The "char" part is an integer representing the position of the source character in the "line" from which the node was created; character postions are numbered from left to right starting at 1. If a node has no equivalence in the source file, "line" and "char" are zero. The "source_designator" is retained in the SOURCE attribute of a node which is part of the external representation of a node. The source file attribute need not be part of the internal representation of DIANA. It could either be stored in the file containing the external representation of DIANA or on a separate file. Regardless, there must be a mapping from the node in main memory to the node on the file which contains the attribute. Because of the inefficiencies associated with the mapping, the source file attribute is also part of the internal representation of a node.

There are circumstances in which it is desirable to store more than one source position with a DIANA node. Useful positions are the left most character of a construct, the rightmost character of a construct, and a "middle" character of a construct (e.g., the position of the infix operator for an expression). Therefore, the DIANA source_position attribute is viewed as a list of "source_designator"s

There is the possibility that a source file is included more than once in a program and it is desirable to be able to map back to the source uniquely.

Reference to the included file in this case is not sufficient, it is necessary to also know the complete qualification of included files starting with the compilation unit containing the first INCLUDE. The qualification is effectively maintained in the Source Table. An entry in the Source Table consists of the database name of a source file (input file and INCLUDE files) and a line count base, i.e., the value of the line counter when the first line of the source file is processed. (Lines are numbered sequentially starting at 1; lines of INCLUDEd files are number sequentially starting with the line number of the INCLUDE.) The base is used to relate line numbers generated by the compiler back to the line in the source file. Instead of storing a database name in the "source_file" of a "source designator", the index of the entry for the source file in the Source Table is stored instead. The Source Table is save in an IL file as part of the IL.

For generic instantiations, there is no source text associated with the expansion. Therefore, the source file attribute for each DIANA node in the expansion references the generic instantiation.

E.3 Internal Representation

There are many kinds of DIANA nodes. Different kinds of nodes are represented as variants of a single record type, NODE, which contains the record component, NODE_KIND, as an enumeration discriminant. NODE is declared with default initial values for the discriminants. This permits variables to be declared without a discriminant constraint; such variables are unconstrained and can be assigned any record value. Node attributes are represented as fields in NODE. Attributes contained in the fixed part of the record are SOURCE, NODE_KIND and NODE_NAME.

During compilation, the number of DIANA nodes is not known in advance. Therefore, node objects are created dynamically by execution of an allocator; node references are represented as access variables of type NODE. Discriminant values are supplied with the allocator to indicate the kind of node to allocate; the allocated node object is contrained by these values.

The abstract syntax tree (AST) is represented by nodes whose NODE_KIND attribute has the value AST_NODE. Attributes in the variant part of an AST node consists of an operation, a node reference to a symbol table entry (a DIANA DEF_ID and its attributes are stored as an entry in the symbol table; all references to the DEF_ID in the AST are replaced by references to the symbol table entry), a node reference to the first subnode of a sequence of subnodes, and a node reference to a sibling. Depending on the operation, the number of subnodes may be fixed or variable in length. Therefore, the subnodes are linked together via the SIBLING field of the AST node; a pointer to the first sibling is contained in the SUBNODES field of the parent node. Each AST node which is not a root is referenced as a subnode (either by the SUBNODES or SIBLING field) in exactly one AST node.

There are many kinds of symbol table nodes each of which is distinguished by a unique NODE_KIND value. How these nodes are linked together depends on the organization of the symbol table (cf. Appendix D).

E.4 External Representation

E.4.1 Naming Nodes

When outputing a compilation unit's DIANA in its external representation, internal node references (access values) must be replaced with an external node reference. Thus, each node must be uniquely named. Since each compilation unit's DIANA is stored separately and because of cross-compilation node references (for example, references to symbol table nodes for non-local types or objects), it is necessary to uniquely identify the compilation unit and the node within the compilation unit. An external node reference is uniquely specified by the pair (compilation_unit, node_label) which is collectively called the "node_name". The "compilation_unit" is a unique positive integer starting at 1. The association of the number of a compilation unit to the file containing the corresponding DIANA is maintained in a table within the library file. The "node_label" is a unique integer which starts at 1.

The "node_name" is retained in the NODE_NAME attribute of a node which is part of the external representation of a node. It is not an attribute in the internal representation. "Node_*name*"*s are generated* when the DIANA is written out to the file, i.e., a pass is first made over the DIANA assigning "node_label"s, then the IL is output to the IL file. The "node_name" does not include the database name of the library file since the IL file is not known outside the context of the library file (i.e., when referencing the IL file for a compilation unit it is necessary to specify the library file which contains the compilation unit).

E.4.2 Text versus Binary Representation

Nodes can be represented externally as text or as binary. Textual representation has an advantage; it can be directly printed or viewed on a CRT. However, converting it to an internal representation requires converting the text to binary and then storing the binary in the fields of a node object; the first step can be eliminated, if the nodes are stored in binary form. Therefore, nodes are represented externally as binary. The nodes are written to a file of records (using the high-level I/O package) where each record is of type EXT_NODE. The type EXT_NODE is similar to the type NODE except access types are replaced by a type representing an external node reference, i.e., node_name. Each field of a node (created by execution of an allocator) is copied to the corresponding field of a variable of type EXT_NODE. Fields of the node containing internal node references (i.e., access values) are mapped to an external node reference (i.e., node name). The node name is obtained from the NODE_NAME field of the node pointed to by the access value.

For an AST node, it is not necessary to map internal node references to other AST nodes external node references if the external representation contains the node's degree (i.e., map SUBNODES and SIBLING access values if the number of subnodes of a node is contained in the external representation). Given the preorder walk of the AST and the degree of each AST node, the internal representation of the AST can be constructed with the internal node references to other AST nodes being implicit. However, the mapping is performed since it may be required by other processing programs. For example, a program to print the AST would not have to construct the internal representation of the AST, but could process the nodes one at a time and print it. Or the information could be used to verify the correct node was being processed. To facilitate constructing the internal representation of the AST, the number of subnodes of an AST node is retained in the DEGREE attribute of the external representation.

E.4.3 Representation of DIANA on Files

The DIANA for a compilation unit is stored in its external representation on a random access file. The DIANA nodes appear in a specific sequence in the file. The IL consists of two physically separate parts: DIANA nodes representing the abstract syntax tree and nodes representing the symbol table. (The DEF-ID's are saved as symbol table nodes. The symbol table constructed by the front-end is saved to expedite processing later by other tools requiring it, including the separate compilation facility of the front end.) A symbol table (cf. Appendix D) consists of three parts: a table of declared entities, a table of name nodes (names are kept separate from their associated declared entity because they can be of arbitrary length), and a table of literal nodes. A compilation unit may contain one or more packages or subprograms. A separate symbol table is maintained for the compilation unit (which is needed for a separate compilation) and for each embedded package, subprogram, task and block. These symbol tables are required by the source level debugger.

The IL (AST and symbol table) for the compilation unit and each embedded package and subprogram are written to the file separately; first the abstract syntax tree nodes, then the symbol table nodes. This permits each piece to be separately processed by the code optimizer and code generator. As each package and subprogram is compiled, its IL is written out. The IL for the compilation unit is written out last. Therefore, the order of the IL is from the most embedded package/subprogram to the outermost package/subprogram. A symbol table is converted into a linear sequence determined by its internal organization, e.g., whether it is a balanced tree or hashed, and the ability to reconstruct the internal organization easily from the linear sequence. An abstract syntax tree is converted into a linear sequence corresponding to a preorder walk of the abstract syntax tree, i.e., visit the root, then the subnodes from left to right. Preorder is selected so declarations appear before body nodes.

The first record of the random access file contains a dictionary giving a map of the IL for the compilation unit and the embedded packages, subprograms, tasks and blocks, i.e., the starting and ending record numbers within the file for each AST and symbol table, and the range of node_labels.

The IL for a generic declaration is delineated so it can be extracted and used in the expansion of generic instantiations. The record also contains the database name of the library file of which the compilation unit is a member, the integer index of the compilation unit's entry in the library file (cf. Appendix F), and the Source Table.

E.4.4 Transforming DIANA on a File to the Internal Representation

For a tool to process the IL of a compilation unit, it must transform the external representation to the internal representation, i.e., construct the internal representation of the abstract syntax tree and the symbol table(s). Requisite symbol tables are transformed first, then the AST is constructed. Constructing the AST or symbol table consists of obtaining a node in its external representation, transforming it to an internal representation, and linking the node into the (partially constructed) data structure. A node's external representation (a binary value) is read from a file into a variable of type EXT_NODE and moved into a new node of the proper NODE KIND. The type of the new node depends on the tool processing the IL, i.e., attributes required by, for example, the code optimizer and the code generator are different. The type supports only the attributes required. The node kind is obtained from the NODE_KIND field of the variable. Based on the NODE_KIND, an allocator is executed with the proper discriminant values and fields of the variable are copied one at a time to the corresponding fields of the allocated node object. Those fields (attributes) not essential to the processing program are not copied. Fields of the variable that contain external node references are mapped to internal node references (i.e., access values) (see below).

E.4.4.1 Constructing a Symbol Table

Certain symbol table nodes contain external node references to other symbol table nodes, e.g., from a symbol entry to an entry in the name table, or links dependent on the lookup algorithm. These external node reference must be mapped to internal node references. Also, certain symbol table nodes contain external node references to AST nodes which must be mapped to internal node references, e.g., a symbol table node for a package symbol would reference AST nodes for the package specification and package body. This presents a problem; AST nodes are loaded after symbol table nodes. Both mappings are handled by use of a node_name dictionary.

An entry in the dictionary consists of a node_name, the kind of node (symbol table or AST), a mark indicating whether the node is loaded or not, and a pointer whose value depends on the mark's value. All symbol table nodes referencing a node_name of a node not loaded are linked together via the node field containing the internal node reference, and the pointer in the dictionary entry for the reference node name points to the head of this list. When a node with node_name is loaded, all nodes on the list of unsatisfied references have their link field set to point to the loaded node. The pointer in the dictionary is changed to point to the loaded node and the entry marked as loaded. If the node_name is a symbol table node in another symbol table (e.g., to a type), this entry is already loaded and the appropriate symbol table is searched for an entry with node_name (recall node name is kept as part of the internal representation of a node). A list of all symbol tables loaded and their location is maintained; the appropriate one is selected via the "compilation unit" part of the node name. Searching the symbol table for the node_name is more efficient spacewise than retaining the symbol table node_name dictionary for each symbol table loaded.

Once the symbol table for a compilation unit is loaded or all symbol tables of embedded packages and subprograms in a compilation unit are loaded, the dictionary need be retained only if AST nodes of the same compilation unit are to be loaded (cf. Section E.4.4.2). The dictionary is used to resolve external node references to/from symbol table nodes from/to AST nodes. The only node name entries in the dictionary marked as unloaded are for references to AST nodes. Depending on the processing to be performed, it may be necessary to load these AST nodes and their subtrees in which case the dictionary is required.

E.4.4.2 Constructing an AST

Given the preorder for the AST and the degree of a node, it is a simple matter to construct the internal representation of the AST. The algorithm processes one node at a time retaining on a stack a pointer to the nodes whose subnodes have not been processed yet and a count of the number of subnodes still required. When the count goes to zero the pointer is removed from the stack and the node is linked to the node referenced by the pointer on top of the stack. Likewise, a node with degree zero is linked to the node referenced by the pointer on top of the stack. Each time a node is linked, the count of the top element on the stack is decremented by 1.

The external node references to other AST nodes comprising the AST being loaded (i.e., the node references in the SUBNODES and SIBLING fields) are not mapped to internal node references (i.e., access values) because the internal node reference is implicit in the construction algorithm. The internal node reference is assigned to the element in the subnode sequence determined by the count on top of the stack (a count equal to the node's degree indicates the first element and a count equal to one indicates the last element).

The external node references to symbol table nodes are mapped to access values by either looking up the external node reference in the node_name dictionary of the symbol table just loaded or searching the appropriate symbol table for an entry with node_name.

As each AST node is loaded and linked into the AST, its node_name is looked up in the node_name dictionary. If not found, nothing happens. If found, all nodes on the list of unsatisfied references have their link field set to point to the loaded AST node. The dictionary entry is not updated to where the node is loaded since node_names are unique.

There are no external node references between AST nodes in different IL files for "updates to previously compiled units are forbidden in DIANA" [GO081, Section 3.4.3.2]. However, such node references may be temporarily established during the processing of the AST.

E.5 Visible Representation

The visible representation of DIANA is its text form. There are a number of reasons for having such a representation. First, it is a transportable representation between computers. Second, it is useful as a debugging aid, particularly for the compiler. Finally, it leaves the decision of the internal representation to the processing programs. The definition of DIANA [GOO81] defines a visible representation as does the IDL [NES81] formal description. *This visible representation shall be used.* It shall be produced by a program that processes the external representation of DIANA on an IL file.

E.6 Derivation of Dialects

The DIANA output by the compiler is viewed as the common interface for all tools that use DIANA. This dialect contains all the information about a program. Therefore, a tool can derive any additional attributes needed for its processing. For example, the code optimizer can derive the flowgraph or procedure call graph from the common dialect. There must be a mechanism to derive a dialect from the common dialect, i.e., to delete attributes not needed and define new ones that are. Also, it must be possible to write out the new dialect in its external representation (with attributes that are needed only for the program itself deleted) and to read it in. This permits the new dialect to be used as an interface to other tools.

A mechanism to derive dialects is a utility program that accepts a definition of the input and output dialects (i.e., a definition of the external representation of the DIANA input and output, a definition of the internal form, together with a definition of the mapping between the external representation of the input/output dialects and the internal form). The input and output dialects may be the same. The output of this program is an Ada package containing the data structure definitions for the input/output dialects and the internal form, and a set of procedures for reading the input external form into the internal data structure and writing the internal form to the output external form. Such a utility program could generate a package to input the common dialect, transform it into an internal representation, and write out a new dialect in its external representation. A package could also be generated to read/write a given dialect from/to its external representation to/from its internal form.

IDL [NES81], which is used to describe DIANA, *shall be used to specify the data structures for different dialects. A program shall be written that converts an IDL specification into an implementation of an Ada package with corresponding reader/writer routines.* This effort shall utilize the work being done at Carnegie-Mellon University where similar programs for different languages are being developed.

APPENDIX F

THE LIBRARY FILE UTILITY

F.1 Introduction

Information about a program required by the Ada separate compilation facility and by AIE tools (e.g., the source level debugger, the program binder, and the command interpreter) is maintained in a library file. The library file utility provides, via subprograms, the primitives to manipulate the contents of a library file.

F.2 The Library File

The library file is a database object. The database system manages concurrent access to the library file; multiple tools may be reading a library file, but only one tool may be writing to a library file. The library file is a separate database for maintaining the compilation state of a program or family of programs.

It is the function of a tool accessing a library file to maintain its consistency, i.e., a tool modifying a library file must do so in a consistent manner. A library file is modified by making a memory copy, modifying the copy, and replacing the library file with the memory copy. While the tool is modifying a library file, it has exclusive access to it. As an example, consider the compiler. At the start of a compilation, the compiler makes a memory copy of the library file. This memory file is modified during the compilation to reflect the changes in the compilation states of the compilation units processed. When the compilation is finished, the compiler decides whether the compilation was successful based on the severity of the errors detected. If the compilation was successful, it replaces the library file with the new updated memory copy. No other compilations may occur in parallel using the same library file, for exclusive access to a library file is granted to only one instance of the compiler.

Within the library file the hierarchical dependencies (genealogy) of the compilation units are maintained. This is, in general, a forest structure. The roots of the forest are library units; each tree consists of subunits contained in the library file. There may be a number of library units (subprograms) that can serve as the main program. The main program is ascertained by the fact it is a library unit and is specified at the time the program is bound; there is no MAIN pragma to indicate a main program. Thus, the library file can contain the compilation unit(s) for a single program or a family of programs. The main program is specified to the program binder which binds the correct compilation units into an executable program.

A library file consists of information describing the library file, a table of compilation unit names, and an entry for each compiled compilation unit.

F.2.1 The Library File Descriptor

The library file descriptor contains global information used by the library file utility to maintain it. This information consists of:

* The database name of the library file. This is needed to recreate the pathname of the library file. It is used, for example, by the compiler when forming default pathnames.

* Signature. An integer value used in the construction of default pathnames; it is concatenated to the file category. The value is incremented each time a default pathname is created.

* The number of compilation units in the library file. When a compilation unit is compiled for the first time, this number is incremented and associated with the compilation. The associated number, known as the compilation unit's configuration index, uniquely identifies the compilation unit within a library file.

It is not necessary to retain references to the roots of the forest structure. These references can be derived by searching the compilation unit name table for library units.

F.2.2 Compilation Unit Name Table

The compilation unit name table provides a map from the name of a compilation unit to the entry for the compilation unit. There is one entry in the table for each compilation unit which consists of:

* The compilation unit's name. The name is a string; it is the name of the compilation unit as it appeared in the source text. It is used, for example, by the compiler when forming default pathnames.

* The nature of the compilation unit:

- A mark indicating whether the compilation unit is a library unit or a subunit.
- A mark indicating whether the compilation unit is for a subprogram or package declaration, or for a subprogram, package, or task body.
- A mark indicating whether the compilation unit is generic, and, if so, whether it is a generic declaration or a generic

instantiation. If it is generic, the previous mark indicates whether it is a subprogram or package.

* The configuration index of the compilation unit's specification. For a subprogram with no separate specification, the index is zero.

* The configuration index of the compilation unit's body. For a package with no separate body, the index is zero.

* A reference to the compilation unit's descriptor. The descriptor contains all of the relevant information associated with the compilation unit.

The forest structure is maintained in the compilation unit name table. Each compilation unit that is a library unit is the root of a tree in the forest. Entries of subunits of a compilation unit are linked together to form the genealogy of the compilation unit. The genealogy of a compilation unit can be used to obtain a list of all its subunits.

Either the configuration index or the name of a compilation unit can be used to look up the entry for a compilation unit. If the compilation unit is a subunit, its name may not be unique; only names of subunits of a library unit need be unique. In this case, the name of the subunit's ancestor that is a library unit must be supplied. The ancestor library unit is looked up first, then its genealogy is traversed to locate the subunit.

F.2.3 The Compilation Unit Descriptor

The compilation unit descriptor contains information specific to a compilation unit. Not all information associated with a compilation unit is maintained in the descriptor, in particular, information which can be derived by other means. For example, the major outputs of each compiler pass, i.e., an IL file, an optimized IL file, and an object module form nodes in the derivation tree for the compilation unit. This derivation tree is maintained in the database instead of the library file via derivation attributes. The derivation attributes consist of Input_File, Processor_Name, and Control_File. The derivation attribute Input_File is the database name of the compilation unit processed by the compiler pass. The attribute Processor_Name is the database name of the compiler used to process the compilation unit. The attribute Control_File is the database name of the compiler control file containing the input parameters to the compiler pass. These derivation attributes are set by the compiler. Other output files of a compiler pass, such as error files, statement map, compiler statistics, symbol map, and type map, are associated with the major output of that pass via database relations. These relationships are set by the compiler. As an example, consider the determination of the error files producted during a compilation. Given the database name of the object file, the derivation path in the derivation tree can be determined via the derivation attributes. The derivation path consists of the major outputs of each compiler pass. The database names of the error files generated by each pass are obtained by examining the appropriate database relation for each of the major outputs.

A compilation unit descriptor consists of:

- Compilation unit's name table reference: a reference back to the entry for the compilation unit in the compilation unit name table.
- Compilation time and date stamp: records the order of compilations. It is used to check for required recompilations (see INCLUDE list below).
- Compile indicator: a mark that indicates whether the compilation unit has been compiled or needs to be recompiled. When stubs are met for the first time, a compilation unit descriptor is created for the subunit and this mark in the descriptor is set to indicate the subunit needs to be compiled. Once a compilation unit has been compiled and it is determined it needs to be recompiled, the mark is set accordingly.
- WITH list: a list of library units appearing in WITH clauses of a context specification. An entry in the list consists of the database name of the library file containing the library unit and the configuration index of the library unit within the library file. The database name of the library file is needed when the library unit is from another program. The list is needed in the determination of the correct compilation order, i.e., what has been compiled, and in the creation of the compilation context for a compilation unit (the WITH list for a subprogram, package, or generic declaration is inherited by the corresponding subprogram or package body, and the WITH list for a library unit is inherited by its subunits). The list also defines the dependence relations between library units which are used in the determination of a consistent elaboration order of these units.
- USE list: a list of packages appearing in USE clauses of a context specification. An entry in the list consists of the name of a package, the configuration index of the compilation unit containing the package, and the database name of the library file containing the compilation unit. The list is needed in the creation of the compilation context for a compilation unit; USE clauses are inherited in the same manner as WITH clauses. The list is also used in the determination of compilation units that need to be recompiled. When a package is recompiled, then all compilation units that use it must be recompiled.
- SEPARATE list: a list of all ancestors of a subunit starting with the ancestor library unit. An entry in the list consists of the configuration index of each ancestor compilation unit within the library file. The list is empty for library units. The list is used in the creation of the compilation context of the subunit. It is also used by the compiler in the formation of default pathanmes.
- Package reference list: a list of all packages appearing in USE clauses within the compilation unit. An entry in the list consists

of the name of a package, the configuration index of the compilation unit containing the package, and the database name of the library file containing the compilation unit. The list is used in the determination of the elaboration order of library units that are package bodies.

* INCLUDE list: a list of all text files specified in INCLUDE pragmas within the compilation unit. An entry in the list consists of the database name of the text file. The list is used by the program binder to determine if any units being bound need to be recompiled, i.e., to check that included files in the unit were not modified by a text editor after the compilation unit was compiled. The check involves comparing the compilation time and date stamp of the compilation unit with the edit time and date stamp of the included file.

* Source file reference: the database name of the source file containing the compilation file. It is used to verify the input to the back end of the compiler was derived from a compilation unit in the library file. This database name is checked against the database name of the library unit from which the input was derived.

* Subprogram declare list: a list of all subprograms declared within the compilation unit. The list is used to construct a call graph in the determination of whether an inline subprogram can be expanded. An entry in the list consists of: 1) a spec_body indicator which states whether the specification or body of the subprogram appeared in a declarative part; 2) the name of the subprogram; and 3) a subprograms called list. The spec_body indicator is used in the determination of compilation units that need to be recompiled when the compilation unit containing the declaration of the subprogram is recompiled. In this case, compilation units containing calls to the subprogram need only be recompiled if the body of the subprogram was recompiled or the specification was recompiled but remained unchanged. The name of the subprogram is the node name of the symbol table node for the subprogram; the node name uniquely identifies the declared subprogram.

* The subprograms called list is a list of all subprograms called by a subprogram declared within the compilation unit. The list is used to construct a call graph for inline expansion and in the determination of compilation units that need to be recompiled (see spec_body indicator above). An entry in the list is: 1) an inline indicator which states whether the call was expanded inline or out-of-line; 2) a reference to the compilation unit containing the subprogram called; and 3) the name of the subprogram called. The inline indicator is used in the determination of whether to expand an inline subprogram. The compilation unit reference consists of the database name of the library file containing the compilation unit and the configuration index of the compilation unit within the library file. The name of the subprogram called is the node name of the symbol table node for the subprogram; the node name uniquely

identifies the subprogram called. The list is also used in the determination of the elaboration order of library units that are package bodies.

- Generic instantiation list: a list of all generic instantiations within the compilation unit. An entry in the list is the node name of the DIANA AST generic instantiation node. This list is used by the program binder in performing generic optimization.

F.3 Functional Capabilities

The primitive functional capabilities supported by the library file utility are:

- Create library file: a library file is created and initialized. Initialization consists of initializing the library file descriptor and entering the compilation unit name table to the predefined environment STANDARD.
- Include members from another library.
- Delete members in a library file.
- Create an entry for a compilation unit
- Access information recorded in a library descriptor, in a compilation units descriptor, and in an entry of the compilation unit name table. This permits a compilation unit's compilation state to be examined and/or modified.
- Open and close a library file.
- Load a copy of the library file into memory.
- Update a library file with a memory copy.
- Assign the next signature(s). Used in the automatic generation of pathnames.

F.4 Library File Interface

To isolate the implementation of the primitives provided by the library file utility from the tool which uses them, the library file utility is implemented as a package. The structure of the library file is private and is operated on by the primitives (subprograms). Any additional functional capabilities required by a tool, such as determination of a recompilation list, checking if all units have been compiled, displaying the forest structure, displaying the roots of the forest, listing all the compilation units, listing all subunits of a compilation unit, or listing the derivation tree of a compilation unit, must be built in terms of the primitives. If the primitives provided by the library file utility are changed, or the library file utility package is recompiled, every tool using it must be recompiled. This can be prevented by moving the new functional capabilities out of the tool into a package known as a library file interface (cf. Figure F-1). Each interface subprogram would implement its functional capability utilizing the primitive capabilities provided by the library file utility package. Changes in the implementation of the library file utility affects only the library file interface; the tool need not be recompiled.

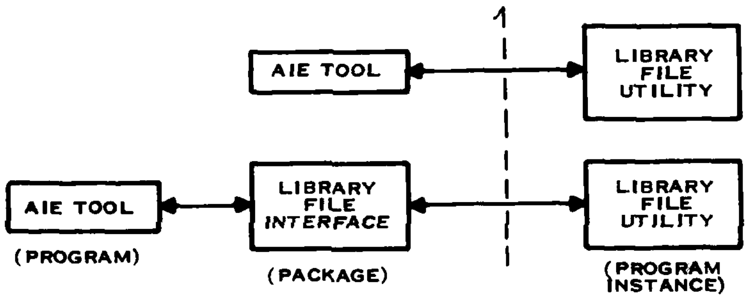


Figure F-1 Interface with the Library File Utility

APPENDIX G

REFERENCES

[AHO77] Aho, A.V. and J.D. Ullman, Principles' of Compiler Design, Addison-Wesley Publishing Company, Reading, MA (1977).

[ALL76] Allen, F.E., A Program Data Flow Analysis Procedure, CACM 19, 3 (March 1976), 137-147.

[BAR80] Barnes, J.G.P., An Overview of Ada, Software-Practice and Experience, 10 (November 1980), 851-887.

[BAR79] Barrett, W.A. and J.D. Couch, Compiler Construction: Theory and Practice, SRA (1979).

[BAT76] Bates, D. (editor), Program Optimization, Infotech State of the Art Report, Infotech International Limited (1976).

[BEL80] Belmont, P.A., Type Resolution in Ada: An Implementation Report, SIGPLAN Notices, 15,11 (November 1980), 57-61.

[BRO80A] Brosgol, B.M., et.al., TCOL Ada: Revised Report on an Intermediate Representation for the Preliminary Ada Language, Carnegie-Mellon University, Computer Science Department (February 1980).

[BRO80B] Brosgol, B.M., TCOL-Ada and the "Middle End" of the PQCC Ada Compiler, SIGPLAN Notices, 15,11 (November 1980), 101-112.

[CAT77] Cattell, R.G., A Survey and Critique of Some Models of Code Generation, Carnegie-Mellon University, Computer Science Department (November 1977).

[CAT78] Cattell, R.G., Formalization and Automatic Derivation of Code Generation, PhD Thesis, Carnegie-Mellon University (April 1978).

[CAT79] Cattell, R.G., et. al., Code Generation in a Machine-independent Compiler, SIGPLAN Notices, 14,8 (August 1979), 65-75.

[CAT80] Cattell, R.R., Automatic Derivation of Code Generators from Machine Descriptions, ACM Transactions on Programming Languages and Systems, 2,2 (April 1980), 173-190.

[CII80] CII Honeywell Bull, Formal Definition of the Ada Programming Language, Louveciennes, France (November 1980).

[DAU79A] Dausmann, M., et. al., Notes on TCOL, Universitat Karlsruhe, West Germany (October 1979).

[DAU79B] Dausmann, M., et. al., AIDA: An Intermediate Representation of Ada Programs - Global Design, Universitat Karlsruhe, West Germany (November 1979).

[DAU79C] Dausmann, M., et. al., AIDA: An Intermediate Representation of Ada Programs, Universitat Karlsruhe, West Germany (November 1979).

[DAU80A] Dausmann, M., et. al., AIDA: An Informal Introduction, Universitat Karlsruhe, West Germany (February 1980).

[DAU80B] Dausmann, M., et. al., AIDA: Reference Manual (Preliminary Draft), Universitat Karlsruhe, West Germany (February 1980).

[DAU80C] Dausmann, M., et. al., Command Interpreter of the Library-User-System (User Information), Universitat Karlsruhe, West Germany (July 1980).

[DAU80D] Dausmann, M., et. al., AIDA: An Informal Introduction (Draft), Universitat Karlsruhe, West Germany (November 1980).

[DAU80E] Dausmann, M., et. al., AIDA: Reference Manual (Draft), Universitat Karlsruhe, West Germany (November 1980).

[DAU80F] Dausmann, M., et. al., SEPAREE: A Separate Compilation System for Ada (Draft), Universitat Karlsruhe, West Germany (November 1980).

[DAV80] Davidson, J.W. and C.W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, ACM Transactions on Programming Languages and Systems, 2,2 (April 1980), 191-202.

[DED80] Dedourek, J.M. and U.G. Gujar, Scanner Design, Software-Practice and Experience, 10 (December 1980), 959-972.

[FAI80] Faiman, R.N. and A.A. Kortesoja, An Optimizing Pascal Compiler, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6 (November 1980), 512-518.

[GAN80] Ganzinger, H. and K. Ripken, Operator Identification in Ada: Formal Specification, Complexity, and Concrete Implementation, SIGPLAN Notices, 15,2 (February 1980), 30-42.

[GES72] Geschke, C.M., Global Program Optimization, PhD Thesis, Carnegie-Mellon University, Computer Science Department (October 1972).

[GOOD80] Goodenough, John B., The Ada Compiler Validation Capability, SIGPLAN Notices, 15,11 (November 1980), 1-8.

[GOO80] Goos, G. and G. Winterstein, Towards a Compiler Front-End for

Ada, SIGPLAN Notices, 15,11 (November 1980), 36-46.

[GOO81] Goos, G. and W. A. Wulf, (editors), Diana Reference Manual, Carnegie-Mellon University and University of Karlsruhe Report (February 1981).

[GRA79A] Graham, S.L., W.N. Joy, and O. Roubine, Hashed Symbol Tables for Languages with Explicit Scope Control, Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices, 14,8 (August 1979), 50-57.

[GRA80] Graham, S.L., Table-Driven Code Generation, Computer (August 1980), 25-34.

[HAR75] Harrison, W.H., A Class of Register Allocation Algorithms, IBM Watson Research Center, Yorktown Heights, NY (March 1975).

[HAR79] Harrison, W.H., A New Strategy for Code Generation-the General-Purpose Optimizing Compiler, IEEE Transactions on Software Engineering, Vol. SE-5, No. 4 (July 1979), 367-373.

[HAR77] Hartmann, A.C., A Concurrent Pascal Compiler for Minicomputers, Springer-Verlag, Berlin (1977).

[HET77] Hecht, M.S., Flow Analysis of Computer Programs, American-Elsevier, New York, NY (1977).

[HIS80] Hisgen, A. et. al., A Runtime Representation for Ada Variables and Types, SIGPLAN Notices, 15,11 (November 1980), 82-90.

[IBM71] PL/I (F) Compiler Program Logic Manual, IBM Corporation, Order No. GY28-6800-5 (December 1971).

[IBM72] FORTRAN IV (H) Compiler Program Logic Manual, IBM Corporation, Order No. GH28-6642-5 (October 1972).

[ICH79] Ichbiah, J.D. et. al., Rationale for the Design of the Ada Programming Language, SIGPLAN Notices, 14,6 (June 1979).

[JAN80] Janas, J.M., A Comment on "Operator Identification in ADA" by Ganzinger and Ripken, SIGPLAN Notices 15,9 (September 1980), 39-43.

[JOH75] Johnsson, R.K., An Approach to Global Register Allocation, PhD Thesis, Carnegie-Mellon University, Computer Science Department (December 1975).

[KOR80] Kornerup, P., et. al., Interpretation and Code Generation Based on Intermediate Languages, Software Practice and Experience, 10,8 (August 1980), 635-658.

[LAM80A] Lamb., David A., Construction of a Peephole Optimizer, Carnegie-Mellon University, Computer Science Department (August 1980).

[LAM80B] Lamb, David A., et.al., The Charrette Ada Compiler, Carnegie-Mellon University, Computer Science Department (October 1980).

[LEB79] LeBanc, R.J. and C.N. Fischer, On Implementing Separate Compilation in Block-Structured Languages, SIGPLAN Notices, 14,8 (August 1979), 139-143.

[LEV79] Leverett, B.W. et.al., An Overview of the Production Quality Compiler-Compiler Project, Carnegie-Mellon University, Computer Science Department (February 1979).

[LEV80] Leverett, B.W. et. al., An Overview of the Production-Quality Compiler-Compiler Project, Computer (August 1980) 38-49.

[MIN79] Mintz, R.J. et. al., The Design of a Global Optimizer, SIGPLAN Notices, 14,8 (August 1979), 226-234.

[MoD] United Kingdom Ministry of Defence, Ada Support System Study: Phase 2 and 3 Reports.

[NES81] Nestor, J. R., W. A. Wulf, and D. Lamb, IDL-Interface Description Language: Formal Description, Carnegie-Mellon University, Computer Science Department (Februrary 81).

[PEM80] Pemberton, S., Comments on an Error-recovery Scheme by Hartmann, Software-Practice and Experience, 10 (1980), 231-240.

[PEN80] Pennello, T., F. DeRemer, and R. Meyers, A Simplified Operator Identification Scheme for Ada, SIGPLAN Notices, 15,7&8 (July-August 1980), 82-87.

[PER80] Persch, G. et. al., Overloading in Preliminary Ada, SIGPLAN Notices, 15,11 (November 1980), 47-56.

[ROS80] Rosenberg, J. et. al., The Charrette Ada Compiler SIGPLAN Notices, 15,11 (November 1980), 72-81.

[RUD79] Rudmik, A. and E.S. Lee, Compiler Design for Efficient Code Generation and Program Optimization, SIGPLAN Notices, 14,8 (August 1979), 127-138.

[SCH73] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, Inc., New York, NY (1973).

[SCH77] Scheifler, R.W., An Analysis of Inline Substitution for a Structured Programming Language, CACM 20,9 (September 1977), 647-654.

[SHE80A] Sherman, M.S. and M.S. Borkan, A Flexible Semantic Analyzer for Ada, SIGPLAN Notices, 15,11 (November 1980), 62-71.

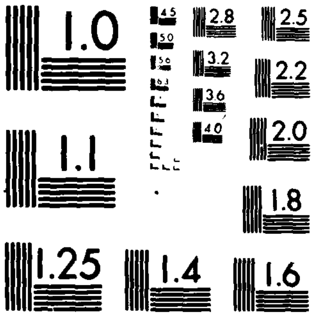
[SHE80B] Sherman, M. et. al., An Ada Code Generator for VAX 11/780 with Unix, SIGPLAN Notices, 15,11 (November 1980), 91-100.

AD-A109 978 TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP F/G 9/2
ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPE--ETC(U)
DEC 81 F30602-80-C-0293

UNCLASSIFIED RADC-TR-81-360-VOL-3 NL

3 of 3
AD
A109978

END
DATE
FILMED
3-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

[SIT79A] Sites, R.L., Machine-Independent Register Allocation, SIGPLAN Notices, 14,8 (August 1979), 221-225.

[SIT79B] Sites, R.L. and D.R. Perkins, Universal P-Code Definition, Version 0.3, Department of Electrical Engineering and Computer Sciences, University of California at San Diego (July 1979).

[WEL78] Welsh, J., Economic Range Checks in Pascal, Software-Practice and Experience, 8 (1978), 85-97.

[WIN80] Winterstein, G. et. al.,The Development of a Compiler Front-End for Preliminary Ada: Overview, University of Karlsruhe (August 1980).

[WUL75] Wulf, W.A. et. al., The Design of an Optimizing Compiler, American-Elsevier, New York, NY (1975).

[WUL80] Wulf, W.A., PQCC: A Machine-Relative Compiler Technology, Carnegie-Mellon University, Computer Science Department (September 1980).

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DATE
FILMED

3-8